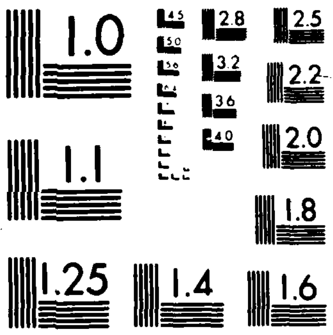
AD-A172 839 DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING PROCESSES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH R A ADAMS 1986 AFIT/CI/NR-86-174D 1/3

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AD-A172 839

DTIC FILE COPY

(1)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 86- 174D	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Distributed Deadlock Detection For Communicating Processes		5. TYPE OF REPORT & PERIOD COVERED ~~THESIS~~/DISSERTATION
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Arthur Adams		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: University of Illinois		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1986
		13. NUMBER OF PAGES 208
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

[illegible]

LYNN E. WOLAVER 22 Sept 86
Dean for Research and Professional Development
AFIT/NR

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

DTIC ELECTE OCT 1 1986 E

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED ...

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

86 10 10 108

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DISTRIBUTED DEADLOCK DETECTION
FOR COMMUNICATING PROCESSES

ABSTRACT

In a distributed system where processes communicate directly, deadlock among a set P of processes occurs when all processes in P are idle waiting for messages from other processes in P in order to start execution, but there are no messages in transit between them. For a process which suspects that it may be deadlocked to determine whether it is indeed deadlocked, it is necessary for it to query other processes. Chandy, Misra, and Haas have proposed a distributed deadlock detection algorithm which uses fixed-length *queries* and *replies*. According to this algorithm, a process which suspects it may be deadlocked initiates a *query computation* by querying each process from which it is waiting to receive a message. If the initiator receives one *reply* for each *query* sent out, then the initiator is deadlocked. Otherwise, if the initiator has not received all *replies* within a timeout period T, it assumes that it is not deadlocked. We present five algorithms which use variable-length *queries* and *replies* to detect deadlock. Instead of using timeout to indicate an absence of deadlock, these algorithms use explicit messages called *informs* to convey the absence of deadlock to the query computation initiator. One algorithm detects a deadlock if it exists when the query computation is initiated. The other algorithms will detect deadlock as deadlock conditions develop. Proofs of correctness are provided, along with a simulation study which compares the performance of the new algorithms with that of the algorithm by Chandy, Misra, and Haas.

86 10 10 108

DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING PROCESSES

208 pages

BY

RICHARD ARTHUR ADAMS
Major, U.S. Air Force

Doctor of Philosophy in Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

1986

Accession For
NTIS [illegible] X
DTIC [illegible]
U[illegible]
Ju[illegible]
By
Di[illegible]
A[illegible]
Dist [illegible]
A-1

QUALITY INSPECTED 4

© Copyright by
Richard Arthur Adams
1986

DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING PROCESSES

BY

RICHARD ARTHUR ADAMS

B.S., Iowa State University of Science and Technology, 1974
M.S., Air Force Institute of Technology, 1979
B.A., University of Maryland, 1983

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING PROCESSES

Richard Arthur Adams, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1986

In a distributed system where processes communicate directly, deadlock among a set P of processes occurs when all processes in P are idle waiting for messages from other processes in P in order to start execution, but there are no messages in transit between them. For a process which suspects that it may be deadlocked to determine whether it is indeed deadlocked, it is necessary for it to query other processes. Chandy, Misra, and Haas have proposed a distributed deadlock detection algorithm which uses fixed-length *queries* and *replies*. According to this algorithm, a process which suspects it may be deadlocked initiates a *query computation* by querying each process from which it is waiting to receive a message. If the initiator receives one *reply* for each *query* sent out, then the initiator is deadlocked. Otherwise, if the initiator has not received all *replies* within a timeout period T, it assumes that it is not deadlocked. In this thesis, five algorithms which use variable-length *queries* and *replies* to detect deadlock are presented. Instead of using timeout to indicate an absence of deadlock, these algorithms use explicit messages called *informs* to convey the absence of deadlock to the initiator of a query computation. One of the algorithms detects a deadlock if it existed when the query computation was initiated. The other algorithms will detect deadlock as deadlock conditions develop. Proofs of correctness are provided for the algorithms, along with a simulation study which compares the performance of the new algorithms with that of the algorithm by Chandy, Misra, and Haas.

DEDICATION

The fear of the Lord is the beginning of knowledge.
Prov. 1:7a

For the Lord gives wisdom;
from his mouth come knowledge and understanding;
he stores up sound wisdom for the upright;
he is a shield to those who walk in integrity,
guarding the paths of justice and
preserving the way of his saints.
Prov. 2:6-8

ACKNOWLEDGEMENTS

I would especially like to thank my thesis advisor, Professor Jane W.-S. Liu, for her guidance, support, and many suggestions when they were needed most. I would also like to thank the other members of my thesis committee: Professors Geneva Belford, H. George Friedman, Duncan Lawrie, and Daniel Reed. Special thanks go to all the graduate students of Prof. Liu for their kindness, advice, and support.

My parents certainly deserve much of the credit, since it was they who always encouraged me to do my best and to pursue such high endeavors. I shall ever be grateful for the continuing love and support of my father. Most of all, I thank my dear wife, Heather, whose love, understanding, and encouragement has made these three years so enjoyable. I thank all of you for your continued patience, support, and understanding that has made it possible for me to complete this thesis.

TABLE OF CONTENTS

CHAPTER 1.

Introduction

In a distributed system where processes communicate directly [Hoar78], deadlock among a set S of processes can occur when all processes in S are idle waiting for messages from other processes in S in order to start execution. When there are also no messages in transit between any pair of processes in S, none of the processes in S will ever receive a message. All processes in S are therefore permanently idle, or deadlocked. For a process which suspects that it may be deadlocked to determine whether it is indeed deadlocked, it is necessary for it to query other processes. Many models of systems in which processes communicate via message exchanges have been developed for the purpose of studying deadlock detection [Brac85, BrTo84, ChMi79, ChMi83, ChMi85, CoLe82, DiSc80, Fran80, GlSh80, Haas81, MiCh82, Misr83, Ober82, SzSh85]. Other models strictly address the problem of resource deadlocks among database transactions [BeOb81, ChMi82, ChMi83, CoEl71, Gold77, Haas81, Holt72, IsMa78, MaIs80, MeMu79, MiMe84, Ober80, SiNa85, Tsai82].

In the model of diffusing computations by Dijkstra and Scholten [DiSc80], every process is ready to receive messages from all other processes at all times. Hence, all processes must be idle waiting for messages in order for termination to occur. Based on this model, Dijkstra and Scholten [DiSc80] and Misra and Chandy [MiCh82] have proposed algorithms to detect termination in diffusing computations.

Chandy, Misra, and Haas [ChMi83, Haas81] have also developed a model, hereafter referred to as the "communication model", which models systems of distributed processes which communicate via messages. Their model is also able to model implementations of Communicating Sequential Processes as put forth by Hoare [Hoar78]. In the communication model, a process 1) is either in the idle state or in the executing state, 2) may wait selectively for messages from some (not necessarily all) other processes, and 3) may, if it is in the execution state, send a message to another process. Every idle process has associated with it a set of processes called its *dependent set*. An idle process will begin to execute whenever it receives a message from any process in its dependent set. While a process is idle, it may not change state or its dependent set. With this model, deadlock exists whenever any subset of processes wait for each other.

Chandy, Misra, and Haas [ChMi83, Haas81] have proposed a distributed deadlock detection algorithm, hereafter referred to as the CMH Algorithm, which uses fixed-length *queries* and *replies*. According to this algorithm, a process which suspects it may be deadlocked initiates a *query computation* by querying all the processes in its dependent set. This process is referred to as the *initiator*. After querying all processes in their respective dependent sets and receiving replies from them, each idle process in the dependent set of the initiator replies to the initiator that the process is idle waiting for messages from other idle processes. An executing process discards any queries received. If the initiator receives one reply for each query sent out, then the initiator is deadlocked. Otherwise, if the initiator has not received all replies within a timeout period T, it assumes that it is not deadlocked. This algorithm does not work correctly when conditions such as network failures, a large backlog of messages, or a process being aborted, etc., prevent any reply from being received within the timeout period T. Since occurrences of these conditions are unavoidable in any distributed system, this shortcoming of the CMH Algorithm is a serious one.

This thesis first discusses the CMH Algorithm based on the communication model [ChMi83, Haas81]. Five algorithms which use variable-length queries to detect deadlock are presented. Instead of using timeout to indicate an absence of deadlock, these algorithms use explicit messages called *informs* to convey the absence of deadlock to the initiator of a query computation. One of the algorithms detects a deadlock if it existed when the query computation was initiated. The other algorithms will detect deadlock as deadlock conditions develop. A simulation study was done to compare their performance with that of the CMH Algorithm. Proofs of correctness are also provided for the algorithms.

This thesis is organized as follows: Chapter 2 describes in detail the communication model used by Chandy, Misra, and Haas to develop their algorithm. Their algorithm will then be presented. Chapter 3 presents the proposed new algorithms for detecting deadlock among communicating processes. Chapter 4 contains proofs of correctness for the algorithms. The results of a simulation study are presented in Chapter 5, and Chapter 6 contains conclusions and recommendations for future study.

CHAPTER 2.

Background, Assumptions, and Definitions

2.1. Introduction

The deadlock detection algorithms developed in this proposal are based on the communication model developed in [ChMi83, Haas81]. It models a system of processes which communicate via messages. It is assumed that the network is reliable, in that the following conditions hold [DiSc80]: (1) All message delays are arbitrary but finite; (2) Messages are transmitted and received correctly. That is, message contents are never corrupted; and (3) Sequenced message delivery is guaranteed. That is, messages sent by process A to process B are received in the order sent. These assumptions are valid in most modern data communication networks [Tane81].

2.2. The Communication Model

According to Hoare's Communicating Sequential Processes, process A can send a message to process B only if process B is waiting to receive a message from process A. In the communication model, a process may be in one of two states, idle or executing. An executing process A may send one or more messages at any time to other processes. When a process changes state from executing to idle, it waits selectively for messages from some processes. This set of processes is called the *dependent set* of the process. An idle process may receive messages only from processes in its dependent set, whereupon its state changes to executing. The dependent set of an idle process does not change. It is further assumed that processes execute reliably in that they do not fail, abort, or otherwise abnormally terminate [FiLy85]. This model can also be used to model CSP

[ChMi83] and resource management problems.

Consider as an example, a set of processes $\{B, C, D\}$ which jointly serve resource management requests. Process A is idle waiting for a resource grant from process B and a grant from either process C or process D. Processes B, C, and D are in the dependent set of process A. When process B sends a message containing a resource grant to process A, process A will execute and determine that it still needs a resource grant from process C or process D. Process A will then become idle and wait for a message from processes C or D. Its dependent set now contains only processes C and D. Now, suppose process C sends a resource grant message to process A. Process A determines that all its requests have been satisfied. It then sends a message to process D to tell process D that it is no longer waiting for a message from it. Process A then proceeds to carry out more computations, determines a new dependent set, and returns to the idle state.

If an idle process will never receive a message from any process in its dependent set, it is permanently idle, or deadlocked. More formally, a nonempty set S of processes is deadlocked [ChMi79] if and only if

(1) All processes in S are idle;

(2) The dependent set of every process in S is a subset of S; and

(3) There are no messages in transit between processes in S.

A set S satisfying these three conditions is referred to as a deadlocked set.

When any of the deadlock detection algorithms discussed in this thesis is used, processes use two different methods for communication. The first is via exchange of *message* as described above. An idle process needs to receive a *message* in order to execute. The other method is called *query computation.* A query computation is carried out by processes in order to determine whether they are deadlocked. To carry out a query computation, *queries* and *replies* are exchanged by processes. Processes may exchange queries and replies while they are in the idle state.

2.3. The CMH Algorithm

According to the CMH Algorithm, an idle process initiates a query computation by sending a query to each process in its dependent set. Information on the query initiator, a sequence number, the sender, and the receiver are contained in the query header. When an idle process receives a query, it will propogate queries to all the processes in its dependent set unless it has already done so. It counts the number of queries sent out. After receiving the same number of matching replies, it will send a reply to the process from which it received the query. If a process is executing, it simply discards the query. When some queries are discarded, the initiator will not receive the required number of replies. Therefore, deadlock will not be declared.

Consider, as a brief example, a system shown as a directed graph in Figure 1. Nodes in this graph represent processes. Executing processes are shown as a double circle. A directed edge from A to B means that process A is waiting for process B, and therefore B is in the dependent set of A. If, for some process C, there is a directed path from A to C, then C is said to be *reachable* from A, and is in the *reachable set* of A. In the example shown in Figure 1, the reachable set of A is $\{B, C, D, E\}$. (Many details have been omitted from this example in order to show, as simply as possible, the main points of the CMH Algorithm.) As can be seen from this graph, the set of processes $\{B, C, D\}$ is deadlocked. Suppose process B initiates a query computation by sending a query to every process in its dependent set (processes C and D). It will then wait for the two corresponding replies. Process D, upon receiving the query from process B, will send a query to process C. Process C, upon receiving the query from process B, will send a query to process D. After receiving the query from process D, process C will send a reply to process D. Similarly, process D, after receiving the query from process C, will send a reply to process C. Both processes will then send a reply to process B. Once process B receives both replies, it will declare itself deadlocked. If process A initiates a query computation, it will not receive a reply matching the query sent to process E. Therefore, process A will not declare itself deadlocked. Suppose that process E later sends a message to process F and then becomes idle, waiting for process C. Process A will then be deadlocked (along with processes B, C, D, and E). However, process A might never learn it is deadlocked without initiating another query computation. It may be more efficient, however, to have process C or D break the deadlock, and for process E to know that it cannot possibly break the deadlock by acting alone.

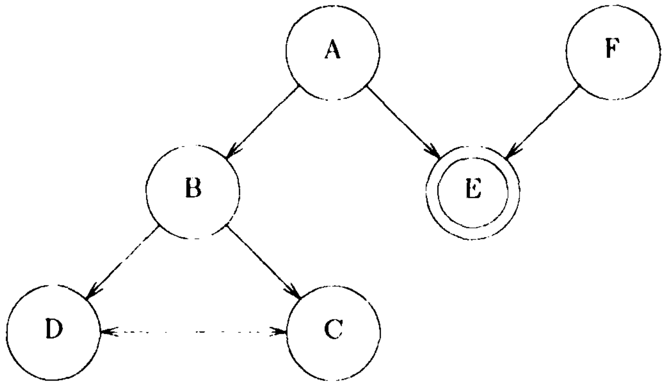


Figure 1
System of Processes as a Directed Graph

More specifically, queries in the CMH Algorithm have the form Q(i, m, j, k) where

i is the process which initiated the query computation (*initiator*) and is the process trying to discover if it is deadlocked or not.

m is a sequence number (counter) which is incremented by one each time process i initiates a new query computation. The sequence number is used to ensure old queries do not interfere with any newer ones initiated by the same process.

j is the process which sends the particular query (*sender*).

k is the process which will receive the query (*receiver*).

Replies to queries have the form R(i, m, j, k) in which fields have the same meaning as in a query. For any given set of values of $\{i, m, j,$ and $k\}$, process P_j will send only one query $Q(i, m, j, k)$ to process P_k. Likewise, process P_k will send only one reply R(i, m, k, j) in response to the query Q(i, m, j, k) from process P_j.

Each process maintains four arrays of local variables. Each array is one-dimensional of length N, where N is the number of processes in the system. The four arrays, as maintained by process P_k, are described below.

LATEST(N)

The entry $LATEST(i)$ holds the largest (most recent) sequence number m for any query Q(i, m, j, k) received by process P_k or query Q(i, m, k, j) sent by process P_k. Note that it is possible for process P_k to initiate a query, in which case $i = k$ in the query Q(i, m, k, j) sent by P_k. For each process, array $LATEST$ is initially set to zeros.

ENGAGER(N)

The entry $ENGAGER(i)$ holds the identity of some process P_j (j not necessarily different from i) which caused $LATEST(i)$ to be set to its current value by sending query Q(i, m, j, k) to process P_k. This array is used to store the information on which process sent the query so that the corresponding reply may be sent back to that process. The initial value of $ENGAGER(i)$ is arbitrary. For the case where $i = k$, $ENGAGER(i)$ is not used by the algorithm.

NUM(N)

The entry $NUM(i)$ holds the total number of queries of the form Q(i, m, k, j) which were sent by process P_k for which no reply has yet been received. When a process queries the processes in its dependent set, $NUM(i)$ will start out containing the number of these processes. As each corresponding reply R(i, m, j, k) is received and is verified ($m = LATEST(i)$), $NUM(i)$ is decremented by one. When $NUM(i)$ becomes zero, all replies have been received. Process P_k then sends reply R(i, m, k, $ENGAGER(i)$) to the process from which it originally received the query. The initial value of $NUM(i)$ is also arbitrary.

WAIT(N)

The entry $WAIT(i)$ shows whether or not process P_k has been continuously idle since $LATEST(i)$ was last updated. If so, then $WAIT(i)$ is true, otherwise it is false. The array $WAIT$ is initialized to false.

The following seven events affect the detection of deadlock. These events are listed together with the actions taken by the process.

(1) An executing process receives a query — An executing process discards all queries received.

(2) An executing process receives a reply — An executing process discards all replies received.

(3) An idle process receives a query — When an idle process P_k receives query Q(i, m, j, k), it performs the following:

```
if $m > LATEST(i)$
then begin
    $LATEST(i) := m$;
    $ENGAGER(i) := j$;
    $WAIT(i) :=$ true;
    for all processes $P_r$ in the dependent set $S$ of $P_k$,
        send query Q($i$, $m$, $k$, $r$)
    $NUM(i) :=$ number of processes in $S$;
end
else if $WAIT(i)$ and $m = LATEST(i)$
    then send reply R($i$, $m$, $k$, $j$) to process $P_j$;
    endif
endif
```

(4) An idle process receives a reply — If an idle process P_k receives a reply R(i, m, j, r) from process P_r, process P_k performs the following:

```
if $m = LATEST(i)$ and $WAIT(i)$
then begin
    $NUM(i) := NUM(i) - 1$;
    if $NUM(i) = 0$
    then if $i = k$
        then declare $P_k$ deadlocked
        else send reply R($i$, $m$, $k$, $j$) to process $P_j$
            where $j = ENGAGER(i)$
        endif
    endif
end
else discard the reply;
endif
```

(5) An idle process receives a message — When an idle process receives a message, it sets $WAIT(i)$ false for all i and begins executing.

(6) An idle process initiates a query — A process, upon becoming idle for time T_1 will initiate a query to determine if it is deadlocked. The parameter T_1 is non-negative and may be set as desired. To initiate a query, process P_i must do the following:

```
begin
    $LATEST(i) := LATEST(i) + 1$;
    $WAIT(i) :=$ true;
    For all processes $P_j$ in the dependent set $S$ of $P_i$,
        send query $Q(i, LATEST(i), i, j)$ to $P_j$
    $NUM(i) :=$ number of processes in $S$;
end;
```

(7) An idle process has waited time T_2 since last initiating a query computation — If an idle process waits for some specified time T_2 and still has not received all the required replies (ie $NUM(i) > 0$), it assumes that it was not deadlocked when the last query computation was initiated. Therefore, the process may continue waiting for some time T_3 before initiating a new query computation. Parameter T_3 is determined in a manner similar to parameter T_1. It is possible, of course, that there is some type of failure in the system. (Recall that message delays may be arbitrary but finite).

Several observations can be made concerning the CMH Algorithm [ChMi83, Haas81]. If a process is deadlocked when it initiates a query computation, it will eventually declare itself deadlocked (assuming, of course, there is no failure in the network or system). Also, if every process initiates a new query computation whenever it becomes idle (or idle for time T_1), at least one process (namely, the last process in the set to become idle) in every deadlocked set will report "deadlock". Note, however, that this algorithm does not guarantee that every process in the deadlocked set discovers or is informed that it is deadlocked. Once a process declares itself deadlocked, it only knows that the processes in its dependent set are also deadlocked. The processes in its deadlocked set must be discovered/notified by other means. If e is the number of communicating pairs in the system, then no more than e queries and replies will be required for a single process to declare itself deadlocked. If N is the number of processes and each process has at most k processes in its dependent set, then $e \leq k * N$. In a system where k is $O(N)$, the CMH Algorithm requires $O(N^2)$ queries and replies for each complete query computation.

CHAPTER 3.

Five Deadlock Detection Algorithms

3.1. Introduction

In this chapter, we describe five new algorithms for detecting deadlock among communicating processes. Each successive algorithm implements a major modification to its predecessor, making it easier to understand how each algorithm works. Algorithm I uses depth-first search techniques instead of breadth-first as in the CMH Algorithm. Algorithm I also introduces the use of the *inform* and the concept of the *query trace list*. The *inform* is introduced to allow an executing process to inform the initiator of a query computation that it is not deadlocked. Algorithm II shows that query computations do not need to be terminated when encountering an executing process. Algorithm III introduces a priority scheme into query computations to reduce the number of *queries* and *replies*. Algorithm IV makes use of relevant information from other query computations, and Algorithm V can be used to find all minimal deadlocked sets.

3.2. Design Goals of the New Algorithms

Several aspects of the CMH Algorithm can be improved. The following goals are established to guide the development of improved algorithms.

(1) All true deadlocks are detected, with no false detections.

(2) Once a process in a deadlocked set declares itself deadlocked, it knows the identity of the other processes in the deadlocked set.

(3) A process which has initiated a query will receive some type of response confirming the presence or absence of deadlock.

(4) The number of *queries* and *replies* should be minimized.

(5) The amount of information passed during query computations should be minimized.

(6) A query computation should identify all minimal deadlocked sets (defined in Section 3.8, page 51) in the reachable set of the initiator. Any process declaring deadlock should also learn if it must take action to break the deadlock.

(7) During the course of the query computation, at least one process in each minimal deadlocked set should also learn the identity of all other processes in its minimal deadlocked set, since at least one process from the minimal deadlocked set will be involved in a scheme to break the deadlock. By identifying all processes in a minimal deadlocked set, an optimizing deadlock breaking scheme could be implemented more easily.

The new algorithms, in performing query computations, use four different types of inter-process communication. These four, called *queries*, *replies*, *cancels*, and *informs*, will be referred to collectively as *query traffic*. *Queries* are sent from an idle process (the *engager* process) to a process in its dependent set (the *target* process) to ask whether or not that process and all processes in its reachable set are idle. A *reply* contains an affirmative response to a *query*, i.e., the process sending the *reply* and all processes in its reachable set are idle. *Cancels* are used to stop a query computation once it has been verified that deadlock condition does not exist. When an executing process receives a *query*, it sends an *inform* to the initiator of the query computation telling it that it is not deadlocked. Since query traffic formats differ for the five algorithms, specific formats will be presented prior to the presentation of each algorithm.

3.3. Additional Assumptions

The communication model as presented in [ChMi83, Haas81] is used with the following three additional assumptions about the system and processes in the system:

(1) Each computer in the network maintains its own clock. There is no single system-wide clock available. Each of the independently-maintained clocks can be synchronized to within a desired tolerance of T_c [LaMe85].

(2) An executing process can be interrupted to process query traffic if it is not already processing query traffic. An executing process alr ady busy processing query traffic queues up all subsequent query traffic awaiting processing.

(3) *Associated with each process* is a priority (not necessarily unique).

The definition of deadlock remains the same as mentioned in Chapter 2.

3.4. Algorithm I

The first deadlock detection algorithm presented, Algorithm I, uses only *queries* and *replies* to carry out deadlock detection. *Cancels* are used to reduce the number of *queries* and *replies* passed once it has been determined that the initiator of a query computation is not deadlocked. *Informs* are used to tell the initiator that it is not deadlocked. After the initiator discovers it is not deadlocked, it may initiate a new query computation at a later time. This algorithm differs from the CMH Algorithm in that a process receiving a *query* will not attempt to simultaneously query every process in its *dependent set*. Rather, *one process in the dependent set* is queried at a time. The corresponding *reply* must be received before the next process in the dependent set is queried. Also, information on which processes have seen the query computation is passed in the *queries* and *replies*.

3.4.1. Query Traffic Formats for Algorithm I

Queries have the form

Q (Sender, Receiver, M, QTL)

In each *query*,

(1) Sender is the process sending the *query*.

(2) Receiver is the process receiving the *query*.

(3) M is the sequence number of the query computation and shows the *query* belongs to the Mth query computation of the initiator.

(4) QTL is a list of processes P_{i_1}, P_{i_2}, $\cdots$, P_{i_j}. A process P_k is in this list if it has been queried during the current query computation. QTL referred to as the *query trace list*. Processes appear in this list in the order they were queried. Hence, the query initiator P_i is the first process in the QTL (i.e. $P_i = P_{i_1}$). The sender of a *query* appears in the query trace list.

Replies have the form

R (Sender, Receiver, M, QTL)

Each field in a *reply* has the same meaning as in a *query*. Both the sender and the receiver of the *reply* appear in the query trace list.

Cancels have the form

C (Sender, Receiver, M, P_{i_1})

In each *cancel*,

(1) Sender and receiver have the same meaning as in a *query*.

(2) M is the sequence number of the query computation being stopped.

(3) P_{i_1} is the initiator of the query computation being stopped.

Informs have the form

I (Sender, Receiver, M, T)

In each *inform*,

(1) Sender and receiver hav‘ the same meaning as in a *query*. Since *informs* will always be sent to the query computation initiator, the receiver is the initiator.

(2) M is the sequence number of the query computation which enables its initiator to discover that it was not deadlocked at the time the query computation was initiated.

(3) T is a timestamp which shows the time the query initiator was verified to be not deadlocked.

3.4.2. Local Variables for Algorithm I

Each process maintains seven arrays of local variables and three scalar variables. Each array is one-dimensional of length N, where N is the number of processes in the system. The seven arrays and three scalar variables, as maintained by process P_k when processing a query computation initiated by P_i, are described below.

LATEST(N)

The entry $LATEST(i)$ holds the largest (most recent) sequence number M for any ***query*** Q (Sender, Receiver, M, QTL) sent or received by process P_k. The initial value of $LATEST(i)$ is zeros.

ENGAGER(N)

The entry $ENGAGER(i)$ holds the identity of some process P_r (r not necessarily different from i) which caused $LATEST(i)$ to be set to its current value by sending a ***query*** Q (P_r, P_k, M, QTL) to process P_k. This array is used to show from which process a ***query*** has been received (***engager*** process) so the corresponding ***reply*** may be sent back to the ***engager***. The initial value of $ENGAGER(i)$ is zeros.

TARGET(N)

The entry $TARGET(i)$ holds the identity of some target process P_r to which process P_k sent ***query*** Q (P_k, P_r, M, QTL) and which has not yet sent back the corresponding ***reply***. A process considers a ***query*** to be ***outstanding*** after it is sent until the corresponding ***reply*** is received or the query computation is cancelled. The initial value of $TARGET(i)$ is also zeros.

DEPENDENT(N)

The entry $DEPENDENT(i)$ is a boolean flag which shows whether or not process P_i is in the dependent set of process P_k. The initial value of $DEPENDENT(i)$ is FALSE. The array $DEPENDENT$ may only be changed when P_k is executing.

REACHABLES(N)

The entry $REACHABLES(i)$ is a boolean flag which shows whether or not process P_i is in the reachable set of process P_k when P_k knows it is deadlocked. The initial value of $REACHABLES(i)$ is FALSE. The array $REACHABLES$ may only be changed when P_k receives a *reply* which causes P_k to declare deadlock.

WAIT(N)

The entry $WAIT(i)$ shows whether or not process P_k has been continuously idle since $LATEST(i)$ was last updated. If so, then $WAIT(i)$ is true, otherwise it is false. The array $WAIT$ is initialized to false.

EDGETYPE(N)

The entry $EDGETYPE(i)$ in the boolean array shows whether the *query* from the query computation initiated by P_i and sent to the process in $TARGET(i)$ is a tree edge (TRUE) or not. $EDGETYPE(i)$ is FALSE if there is no outstanding query or if the query outstanding created a forward, back, or cross edge in the DFS tree [ReNi77]. $EDGETYPE$ is initialized to FALSE.

LASTTIME

The variable $LASTTIME$ stores the time at which the process last confirmed it was not deadlocked. A process may confirm that it is not deadlocked either because it is executing, or because it receives an *inform* from an executing process.

STATE

The variable $STATE$ is a boolean flag which shows if process P_k is in the executing (TRUE) or idle (FALSE) state.

DEADLK

The variable *DEADLK* is a boolean flag which shows if process P_k knows it is deadlocked (TRUE). If P_k does not know, then *DEADLK* is FALSE. *DEADLK* is initialized to FALSE.

3.4.3. Events for Algorithm I

For Algorithm I, the following events are of interest:

(1) An idle process receives a message.

(2) A process receives a *query*.

(3) A process receives a *reply*.

(4) A process receives a *cancel*.

(5) A process receives an *inform*.

(6) A process sends a message.

(7) An executing process changes state to idle.

(8) A process becomes idle for time T_1 since last executing or last receiving an *inform*.

The actions taken by a process when each of these events occurs are described below. It is assumed, but not shown, that all query traffic received is first checked for validity. The variable N contains the number of processes in the system. The pseudo code for Algorithm I is listed in Appendix A.

(1) When an idle process P_r receives a message and begins executing, it takes the following actions:

(a) Changes its state to executing.

(b) Reinitializes the array *WAIT* to FALSE.

(c) For each outstanding ***query***, sends an ***inform*** to the query initiator.

(d) Cancels all outstanding ***queries*** which, when sent, created tree edges (sent to a process not yet in the query trace list).

(e) Removes all evidence of ***queries*** received.

(2) When a process P_r receives a ***query*** Q (P_k, P_r, M, QTL) belonging to a query computation initiated by P_i, it takes the following actions:

(a) If the ***query*** is from a query computation not seen before,

- Updates $LATEST(i)$.

(b) If P_r is executing, then

- Sends an ***inform*** to the query initiator.
- Discards the ***query***.

(c) If P_r is deadlocked, then

- Sets $WAIT(i)$ to TRUE.
- Adds P_r and all members of its reachable set to the query trace list.
- Sends a ***reply*** to the engager process P_k.

(d) If P_r is idle and is not in the query trace list, then

- Sets $WAIT(i)$ to TRUE.
- Adds P_r to the query trace list.
- If there is a process P_l in the dependent set which is not the engager, sends a ***query*** to the first such process. The ***query*** being sent creates either a tree edge (if P_l is not in the query trace list) or a different type of edge (forward,back, or cross edge). Sets $EDGETYPE(i)$ accordingly.
- If the dependent set is empty or contains only the engager, sends a ***reply*** back to process P_k.

(e) If P_r is idle, and P_r is in the query trace list, then

- If P_r has been continuously idle since receiving the query computation the first time ($WAIT(i)$ is TRUE), then sends a ***reply*** to P_k.

- If P_r has not been continuously idle since receiving the query computation the first time ($WAIT(i)$ is FALSE), then sends an *inform* to P_i and discards the *query*.

(3) When a process P_r receives a *reply* R (P_k, P_r, M, QTL) belonging to a query computation initiated by P_i, it takes the following actions:

(a) If P_r is deadlocked

- Adds all members of its reachable set to the query trace list.
- Sends a *reply* to the engager process.

(b) If P_r is idle but does not know it is deadlocked

- If there is a process P_l following P_k in the dependent set which is not the engager, sends a *query* to the first such process. The *query* being sent creates either a tree edge (if P_l is not in the query trace list) or a different type of edge (forward,back, or cross edge). Sets $EDGETYPE(i)$ accordingly.
- Otherwise, if P_r is the query initiator, declares DEADLOCK for P_r and all processes in the query trace list. For each process P_j appearing in the QTL, sets $REACHABLES(j)$ to TRUE.
- Otherwise (P_r is not the initiator and the dependent set has been queried except possibly the engager), sends a *reply* to the engager process.

(4) When a process P_r receives a *cancel* C (P_k, P_r, M, P_i), it takes the following actions:

(a) If P_r has an outstanding *query* with the same initiator and matching sequence number which was sent to a process not in the query trace list (tree edge), sends the *cancel* to this process.

(b) Removes all evidence of the matching *query* (whether or not a *cancel* was sent).

(5) When a process P_r receives an *inform* I (P_k, P_r, M, T), it takes the following actions:

(a) Updates the time that it last knew that it was not deadlocked.

(b) May initiate a new query computation at some later time in order to detect a possible deadlock.

(6) When a process P_k sends a message, no query computation action is required.

(7) When an executing process P_k changes state to idle, it takes the following actions:

 (a) Changes its $STATE$ flag to show it is idle.

 (b) Sets the lasttime that it knew that it was not deadlocked.

(8) When a process P_i becomes idle for time T_1 since becoming idle or since last receiving an *inform*, it initiates a new query computation and sends a *query* to the first process P_r in the dependent set.

3.5. Algorithm II

In Algorithm I, a process which suspects it may be deadlocked initiates a query computation. If an *inform* indicating no deadlock is received and at a later time the process still suspects it is deadlocked, it must initiate another query computation. For some systems, such as interactive information retrieval programs operating on distributed databases, the ability of a process to monitor its deadlock status without having to periodically initiate query computations is desirable. This is especially true of a process which interfaces directly with a user, since a monitoring capability can reduce user anxiety during periods of apparent inactivity. Algorithm II serves this purpose.

Algorithm II also uses only *queries* and *replies* to carry out deadlock detection. *Cancels* and *informs* are used for the same purposes as in Algorithm I, although the logic controlling when to send a *cancel* is different. Unlike Algorithm I, according to Algorithm II an executing process does not discard a *query* after sending an *inform* to the query initiator. Rather, the *query* is held and the query computation may continue once the process finishes executing. Because query computations are not discarded when they encounter executing processes, an executing process can periodically inform idle processes suspecting deadlock that they are not deadlocked. Thus, once an idle process initiates a query computation, it effectively monitors its own deadlock status by periodically receiving *informs* from executing processes in its reachable set.

3.5.1. Query Traffic Formats for Algorithm II

Queries have the form

Q (Sender, Receiver, T, QTL)

In each *query*,

(1) Sender is the process sending the *query*.

(2) Receiver is the process receiving the *query*.

(3) T is a timestamp which shows the time the query initiator was last informed that it was not deadlocked.

(4) QTL is a list of process/sequence number pairs $P_{i_1}, M_{i_1}, P_{i_2}, M_{i_2}, \cdots, P_{i_j}, M_{i_j}$ and is referred to as the *query trace list*. A process P_{i_k} which appears in this sequence has been queried during the current query computation and is assumed to be idle by the sender. M_{i_k} is the sequence number associated with process P_{i_k}. Because query computations are allowed to be restarted by processes other than the query initiator, each process must have its own sequence number. This number shows the number of attempts by the process to complete the query computation which has the same query trace list (to include the same associated sequence numbers). Processes appear in the query trace list in the order they were queried. Thus, P_{i_1} is the query initiator. The sender of a *query* appears in the query trace list.

Replies have the form

R (Sender, Receiver, T, QTL)

Each field in a *reply* has the same meaning as in a *query*. Both the sender and the receiver of the *reply* appear in the query trace list.

Cancels have the form

C (Sender, Receiver, P_{i_1} M_{i_1} ..., P_{k_1} M_k)

In each *cancel*,

(1) Sender and receiver have the same meaning as in a *query*.

(2) QTL is a list of process/sequence number pairs $P_{i_1}, M_{i_1}, P_{i_2}, M_{i_2}, \cdots, P_{i_k}, M_{i_k}$ which make up the query trace list as seen by the initiator of the *cancel*. Thus, the first process P_{i_1} is the initiator of the query computation being stopped, and the last process P_{i_k} is the process which initiated the *cancel*. As a *cancel* is passed from process to process, the query trace list portion of the *cancel* does not expand.

Informs have the form

I (Sender, Receiver, M, T)

In each *inform*,

(1) Sender and receiver have the same meaning as in a *query*.

(2) M is the sequence number M_{i_1} from the query computation which enables its initiator to discover that it was not deadlocked at the time it initiated the query

computation.

(3) T is a timestamp which shows the time the query initiator was verified by an executing process to be not deadlocked.

3.5.2. Local Variables for Algorithm II

Each process maintains eight arrays of local variables and four scalar variables. One of the arrays is two-dimensional, and the other seven arrays are one-dimensional of length N, where N is the number of processes in the system. The eight arrays and four variables, as maintained by process P_k when processing a query computation initiated by P_i, are described below.

LATEST(N,2N)

The array $LATEST$ is two-dimensional and is used to hold copies of query trace lists from *queries* received by process P_k. The entry $LATEST(i,*)$ holds a copy of the query trace list from a *query* received by process P_k which was initiated by process P_i. Thus, $LATEST(i,1)$ holds the first entry in the query trace list, which is simply P_i, the query initiator, and $LATEST(i,2)$ holds the corresponding sequence number M_i. The array $LATEST$ is initialized to zeros.

ENGAGER(N)

The entry $ENGAGER(i)$ holds the identity of some process P_r (r not necessarily different from i) which caused $LATEST(i,*)$ to be set to its current values by sending a *query* $Q(P_r, P_k, T, QTL)$ to process P_k. This array is the same as in Algorithm I.

TIMES(N)

The entry $TIMES(i)$ holds the value of the timestamp T carried in the most recently received query which was initiated by process P_i. Thus, $TIMES(i)$ shows when process P_k thinks process P_i last knew that P_i was not deadlocked. It is possible, however, that more recent *informs* have been sent to process P_i which process P_k does not know about.

TARGET(N)

This array is the same as in Algorithm I.

DEPENDENT(N)

This array is the same as in Algorithm I.

REACHABLES(N)

This array is the same as in Algorithm I.

WAIT(N)

The entry $WAIT(i)$ shows whether or not process P_k has been continuously idle since $LATEST(i, *)$ was last updated. This array is the same as in Algorithm I.

EDGETYPE(N)

This array is the same as in Algorithm I.

LASTTIME

This variable is the same as in Algorithm I.

NEWQRYNUM

The variable $NEWQRYNUM$ is a sequence number (counter) which is incremented by one each time process P_k initiates a new query computation. This number will be used in forming the query trace list.

STATE

This variable is the same as in Algorithm I.

DEADLK

This variable is the same as in Algorithm I.

In the description of the events for Algorithm II (shown below), and in the pseudo code (listed in Appendix B), we use $QTL(2k-1)$ to denote process P_{i_k} in the query trace list, and $QTL(2k)$ to denote the sequence number associated with this process. Thus, $QTL(1)$ denotes the first process P_{i_1} in the QTL, i.e. the initiator of the query computation in question.

3.5.3. Events for Algorithm II

For Algorithm II, the following events are of interest:

(1) An idle process receives a message.

(2) A process receives a *query*.

(3) A process receives a *reply*.

(4) A process receives a *cancel*.

(5) A process receives an *inform*.

(6) A process sends a message.

(7) An executing process changes state to idle.

(8) A process becomes idle for time T_i since last executing or last receiving an *inform*.

The actions taken by a process when each of these events occurs are described below. The variable N contains the number of processes in the system. The pseudo code for Algorithm II is listed in Appendix B.

(1) When an idle process P_r receives a message from process P_k and begins executing, it takes the following actions:

(a) Changes its state to executing.

(b) Reinitializes the array *WAIT* to FALSE.

(c) If P_r initiated a query computation and has a *query* outstanding which created a tree edge, sends a *cancel* to the target process. Regardless of the edgetype, deletes the *query*.

(d) If P_r has any other *queries*, either held or outstanding, sends *informs* to the query computation initiators.

(e) For any outstanding *queries* which formed tree edges, sends *cancels* to the target processes.

(f) Shortens each *query* so that $P_{r,} M_r$ is the last process/sequence number pair in the query trace list.

(2) When a process P_r receives a *query* Q ($P_{k,}$ $P_{r,}$ T, QTL), it takes the following actions:

(a) If P_r is executing and has seen the query computation before i.e. it is in the QTL, it discards the *query* just received.

(b) If P_r is executing and has not seen the query computation before,

- Adds itself (with sequence number 1) to the end of the query trace list, updates the time field T in the *query*, and stores a copy of the *query*.
- Sends an *inform* to the query initiator P_i.

(c) If P_r is deadlocked,

- If P_r is not in the query trace list, adds itself and all members of its reachable set which are not already in the query trace list to the query trace list. The sequence numbers associated with all processes added from the reachable set are 0.

- Sends a *reply* to process P_k.

(d) If P_r is idle, and P_r is in the QTL

- If P_r has been continuously idle since receiving the query computation the first time ($WAIT(i)$ is TRUE), or the sequence number associated with P_r in the QTL is 0, then sends a *reply* to P_k.

- If P_r has not been continuously idle since receiving the query computation the first time ($WAIT(i)$ is FALSE), then the *query* will be discarded during the validity check.

(e) If P_r is idle and has not seen the query computation before

- Sets $WAIT(i)$ to TRUE.

- Adds itself (with sequence number 1) to the QTL.

- If there is a process P_l in the dependent set which is not the engager, sends a *query* to the first such process. The *query* being sent creates either a tree edge (if P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). Sets $EDGETYPE(i)$ accordingly. Updates $LATEST(i, *)$.

- Otherwise, sends a *reply* back to process P_k.

(3) When a process P_r receives a *reply* R (P_k, P_r, T, QTL), it takes the following actions:

(a) If P_r has queried all processes in its dependent set except possibly the engager, and P_r is the query initiator, then

- Declares DEADLOCK for P_r and all processes in the QTL.

- For each process P_j appearing in the QTL, sets $REACHABLES(j)$ to TRUE.

- For each *query* outstanding, sends a *cancel* to the target process.

- Shortens the query trace list of each *query* held by P_r so that P_r is the last process in the query trace list. Increases the sequence number associated with P_r by 1.

- For each *query* now being held by P_r, augments the query trace list to include all processes in the reachable set of P_r. The sequence numbers associated with all processes added from the reachable set are 0.

- Sends *replies* to the engager processes.

(b) If P_r has queried all processes in its dependent set except possibly the engager, and P_r is not the query initiator, then

– Sends a *reply* to the engager process.

(c) Otherwise (there is a process P_l following P_k in the dependent set which is not the engager),

– Sends a *query* to the first such process. The *query* being sent creates either a tree edge (if P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). Sets $EDGETYPE(i)$ accordingly.

(4) When a process P_r receives a *cancel* C (P_k, P_r, QTL), it takes the following actions:

(a) If P_r has an outstanding *query* which created a tree edge and which matches the QTL of the *cancel*, sends a *cancel* to the target process.

(b) Deletes the *query*.

(5) When a process P_r receives an *inform* I (P_k, P_r, M, T), it updates the time that it last knew that it was not deadlocked.

(6) When a process P_k sends a message, no query computation action is required.

(7) When an executing process P_k changes state to idle, it takes the following actions:

(a) Changes its $STATE$ flag to show it is idle.

(b) Sets the lasttime that it knew that it was not deadlocked.

(c) Processes any *query* being held by P_k as though it had just been received, increasing by one the sequence number associated with P_k in the QTL.

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, it initiates a new query computation by sending a *query* to the first process in the dependent set. This creates a tree edge. $EDGETYPE(i)$ is set to TRUE.

3.6. Algorithm III

The third deadlock detection algorithm presented, Algorithm III, differs from Algorithm II in that a priority scheme is used to reduce the number of *queries* and *replies* being passed through the system. According to Algorithm III, processes in the system are assigned priorities. A process may change its assigned priority only when it is executing. The priority of a query computation is equal to the priority of the initiator at the time the query computation is initiated. Thus, the priority of a particular query computation does not change. When a *query* is received, the process receiving the *query* holds this *query* if it has another *query* outstanding which has a higher priority. It processes this *query* if the *query* has a priority at least as high as that of any outstanding *query*. Therefore, the highest priority processes will always have their query computations processed and will be able to monitor their own deadlock status just as in Algorithm II. A lower priority process is not guaranteed to receive *informs* periodically after initiating a query computation, since its query computation may be held by other idle processes. Also, an idle process does not begin to query its dependent set until it has been continuously idle for T_2 time.

Higher priority query computations can cause processing of lower priority query computations to be suspended. Therefore, the possibility of starvation for lower priority query computations exists. For example, consider a set of processes with dependency relationships as shown in Figure 2. The priorities of the processes are as indicated, with 2 being the highest priority. Suppose that processes A and B pass messages back and forth between themselves, thus executing alternately, and that all the other processes are in fact deadlocked. In this example, processes C, D, E, and F should all discover they are deadlocked. However, there is a possibility that A and B prevent C, D, E, and F from ever discovering they are deadlocked by sending higher priority queries (and possibly cancels).

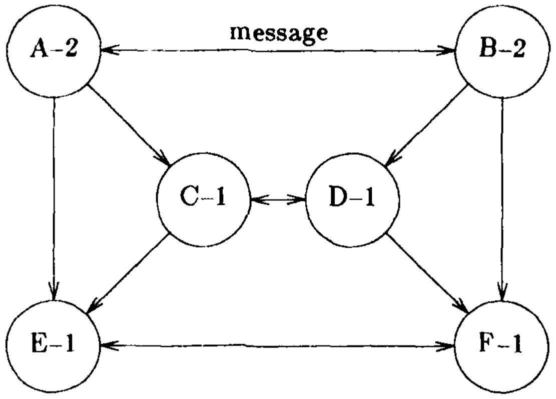


Figure 2
Starvation Example

In order to prevent starvation, an aging scheme is introduced. According to this scheme, each query computation has associated with it an *urgency*. The urgency of a suspended query computation is a *monotone* increasing function of the priority of the query computation and a starvation factor which is computed by a process when processing the query computation. For example, we let the urgency of a query computation be the sum of the priority of the query computation and the integer quotient of the starvation count divided by some user-chosen constant M. To determine how the starvation factor should be increased, we note that it should not be a function simply of the amount of time a query computation is held by a process because it has higher priority query computations outstanding, since the higher priority query computations may have reached an executing process. If this is the case, increasing the urgency of the other suspended query computations with lower priorities tends to defeat the original purpose of the priority system. Without an aging scheme, starvation can only occur when processes send higher priority *queries*, followed by *cancels* and more *queries*. Some processes may reply to these higher priority query computations before receiving the *cancels*. One reliable way to adjust the query computation urgencies is to have the starvation factor based on the number of *cancels* a process receives and the number of *replies* a process sends in response to *queries* from higher priority query computations. How the starvation factor of a suspended query computation is increased will be described later in this section.

3.6.1. Query Traffic Formats for Algorithm III

Queries have the form

Q (Sender, Receiver, U, T, QTL)

In each *query*,

(1) All fields except U have the same meaning as in Algorithm II.

(2) U shows the priority of the query initiator. Although it does not need to be included in the actual *query*, since it could be a function of P_i and/or M_i, it is shown for clarification. Priorities of processes do not have to be unique. If all the priorities of processes are the same, Algorithm III performs much like Algorithm II.

Replies have the form

R (Sender, Receiver, T, QTL)

Each field in a *reply* has the same meaning as in Algorithm II. The priority field is not used in a *reply* since *replies* are never held.

Cancels have the form

C (Sender, Receiver, QTL)

Each field in a *cancel* has the same meaning as in Algorithm II.

Informs have the form

I (Sender, Receiver, M, T)

Each field in an *inform* has the same meaning as in Algorithm II.

3.6.2. Local Variables for Algorithm III

Each process maintains eleven arrays of local variables and five scalar variables. One of the arrays is two-dimensional, and the other ten arrays are one-dimensional of length N, where N is the number of processes in the system. The eleven arrays and five variables, as maintained by process P_k when processing a query computation initiated by process P_i, are described below.

LATEST(N,2N)

This array is the same as in Algorithm II.

ENGAGER(N)

This array is the same as in Algorithm II.

PRIOR(N)

The entry $PRIOR(i)$ holds the priority of process P_i. If the priority of a process is a function of P_i and/or M_i, then this array is not needed.

STARVE(N)

The entry $STARVE(i)$ holds the number of *cancels* P_k received or *replies* P_k sent for engaging *queries* which caused a lower priority query computation initiated by P_i to be suspended by P_k.

TIMES(N)

This array is the same as in Algorithm II.

TARGET(N)

This array is the same as in Algorithm II.

DEPENDENT(N)

This array is the same as in Algorithm II.

REACHABLES(N)

This array is the same as in Algorithm II.

WAIT(N)

This array is the same as in Algorithm II.

EDGETYPE(N)

This array is the same as in Algorithm II.

LASTREP(N)

The entry $LASTREP(i)$ shows which process from the dependent set last sent a *reply* to P_k for a query computation initiated by P_i.

LASTTIME

This variable is the same as in Algorithm II.

NEWQRYNUM

This variable is the same as in Algorithm II.

STATE

This variable is the same as in Algorithm II.

DEADLK

This variable is the same as in Algorithm II.

URGENCY

The variable *URGENCY* may be computed at any time from entries in arrays *PRIOR* and *STARVE*. *URGENCY* is included here for ease of understanding. For some query computation initiated by P_i with priority U, and being held by P_k, the *URGENCY* of the held query computation is the sum of U (found in $PRIOR(i)$), and the integer quotient of $STARVE(i)$ divided by some user-chosen constant M. M normally would be greater than the largest possible number of processes in the system.

In the description of the events for Algorithm III (shown below), and in the code (listed in Appendix C), the one-dimensional array *QTL* is used just as it was for Algorithm II.

3.6.3. Events for Algorithm III

For Algorithm III, the following events are of interest:

(1) An idle process receives a message.

(2) A process receives a *query*.

(3) A process receives a *reply*.

(4) A process receives a *cancel*.

(5) A process receives an *inform*.

(6) A process sends a message.

(7) An executing process changes state to idle.

(8) A process becomes idle for time T_1 since last executing or last receiving an *inform*. This is when the process initiates its own query computation.

(9) A process becomes idle for time T_2 (where $T_2 \leq T_1$) since last executing or last receiving an *inform*. This is when *queries* being held are restarted.

The actions taken by a process when each of these events occurs are described below. It is assumed, but not shown, that all query traffic received is first checked for validity. Events in Algorithm III which are the same as those in Algorithm II are noted as such. The variable N contains the number of processes in the system. The pseudo code for Algorithm III is listed in Appendix C.

(1) When an idle process P_r receives a message and begins executing, the actions required are the same as for Algorithm II except that the array *STARVE* is reset to 0.

(2) When a process P_r receives a *query* Q $(P_{k,} P_{r,} U, T, QTL)$, it takes the following actions:

(a) If P_r is executing and has seen the query computation before (P_r is in the QTL), then

- Discards the *query* just received.

(b) If P_r is executing and has not seen the query computation before, then

- Adds itself to the end of the query trace list using sequence number 1.
- Updates the time field T in the *query*.
- Stores a copy of the *query*.

- Sends an *inform* to the *query* initiator.

(c) If P_r is deadlocked, then

- If P_r is not in the QTL, adds P_r and all members of its reachable set which are not already in the QTL to the QTL. The sequence numbers associated with all processes added from the reachable set are 0.
- Sends a *reply* to process P_k.

(d) If P_r is idle, and P_r is in the QTL, then

- If P_r has been continuously idle since receiving the query computation the first time ($WAIT(i)$ is TRUE), or if P_r has a sequence number 0 in the QTL, then sends a *reply* to P_k. Otherwise the *query* is discarded during the validity check.

(e) If P_r is idle and P_r is not in the QTL, then

- Sets $WAIT(i)$ to TRUE.
- Adds itself (with sequence number 1) to the QTL.
- If the only process in the dependent set which has not seen the query computation is the engager, sends a *reply* back to process P_k.
- Otherwise, there is a process in the dependent set which is not the engager and which has not seen the query computation.
    - Stores the *query*.
    - Sets $STARVE(i)$ to 0.
    - If P_r has been idle for T_2 time since last receiving an *inform* or executing, and if the $URGENCY$ of the *query* just received is at least as high as the highest priority of any *query* currently received, sends a *query* to the first process P_l in the dependent set which is not the engager. The *query* being sent creates either a tree edge (if P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). Sets $EDGETYPE(i)$ accordingly.

(3) When a process P_r receives a *reply* R (P_k, P_r, T, QTL), it takes the following actions:

(a) Updates $LASTREP(i)$ and $LATEST(i,*)$.

(b) If every process in the dependent set has been queried and if P_r is the query initiator, then

- Declares DEADLOCK for P_r and all processes in the QTL.
- For each process P_j appearing in the QTL, sets $REACHABLES(j)$ to TRUE.
- For each *query* outstanding, sends a *cancel* to the target process. Holds the *queries*.
- For each *query* being held, shortens the query trace list so that P_r is the last process in the query trace list. Increases the sequence number associated with P_r by 1.
- For each *query* now being held by P_r, augments the query trace list to include all processes in the reachable set of P_r. The sequence numbers associated with all processes added from the reachable set are 0.
- Sends *replies* to the engager processes.

(c) If P_r has queried all processes in its dependent set except possibly the engager, and P_r is not the query initiator, then

- If the query computation has the highest priority of any received, increases $STARVE(l)$ by one for each l where P_l is the initiator of a query computation currently being held by P_r.
- Sends a *reply* to the engager process.

(d) If P_r has been idle T_2 time since last executing or receiving an *inform*, then

- Determines the highest priority of any *query* received. Selects all *queries* it is has received which have an $URGENCY$ at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. $LASTREP(i)$ shows the progression through the dependent set. Any *queries* being sent create either tree edges (if the target processes P_l are not in the QTL) or different types of edges (forward, back, or cross edges). For each *query* sent, sets $EDGETYPE(i)$ accordingly.

(4) When a process P_r receives a *cancel* C (P_k, P_r, QTL), it takes the following actions:

(a) If P_r is executing and the query computation associated with the *cancel* has the highest priority of any received, increases $STARVE(l)$ by one for each l where P_l is the initiator of a query computation currently being held by P_r.

(b) Processes the *cancel* just as for Algorithm II.

(c) For each of the highest urgency *queries* remaining, if P_k has been idle for time T_2 since last executing or receiving an *inform*, then ensures that the *query* is not being held, since the *cancel* could have stopped the highest urgency query computation.

- Determines the highest priority of any *query* received.
- Selects all *queries* it is has received which have an *URGENCY* at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. $LASTREP(i)$ shows the progression through the dependent set. Any *queries* being sent create either tree edges (if the target processes P_l are not in the QTL) or different types of edges (forward, back, or cross edges). For each *query* sent, sets $EDGETYPE(i)$ accordingly.

(5) When a process P_r receives an ***inform*** I (P_k, P_r, T), the actions required are the same as for Algorithm II.

(6) When a process P_k sends a message, no query computation action is required.

(7) When an executing process P_k changes state to idle, it takes the following actions:

(a) Changes its $STATE$ flag to show it is idle.

(b) Sets the lasttime that it knew that it was not deadlocked.

(c) For each *query* currently being held by P_k, increases by one the sequence number associated with P_k in the QTL. These *queries* will be held until P_k becomes idle for T_2 time since last executing or receiving an *inform*.

(d) If, for any held *query*, P_k does not need to query any processes in its dependent set, then sends a *reply* back to its engager. (This can happen if there is only one process in the dependent set of P_k, and it is its engager).

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle or since receiving the last *inform*, it initiates a new query computation by building a *query* and storing it. If the *URGENCY* of the *query* just created is as high or higher than the highest priority of any received *queries*, then sends the new *query* to the first process in its dependent set. This creates a tree edge. Sets *EDGETYPE* accordingly.

(9) When a process P_i becomes idle for time T_2 (where $T_2 \leq T_1$) since changing state from executing to idle or since receiving the last *inform*, it takes the following actions:

(a) Determines the highest priority of any *query* received.

(b) Selects all *queries* it is has received which have an *URGENCY* at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. $LASTREP(i)$ shows the progression through the dependent set. Any *queries* being sent create either tree edges (if the target processes P_l are not in the QTL) or different types of edges (forward, back, or cross edges). For each *query* sent, sets $EDGETYPE(i)$ accordingly. Lower urgency *queries* are held.

3.7. Algorithm IV

For the other algorithms, sequence numbers in the query trace lists of two different query computations were independent of each other. The fourth deadlock detection algorithm presented, Algorithm IV, differs from Algorithm III in that a process does not increase its sequence number by 1 when initiating a query computation. Rather, sequence numbers show the number of messages accepted by processes and therefore are no longer tied to query computations [ReKa79]. By using this type of sequence number, more information can be conveyed between processes using the query trace list. When processing a query computation, a process P_k can forego querying a member of its dependent set P_r if P_k can determine that P_r is idle and that all messages sent from P_r to P_k have already arrived. This can sometimes be determined by combining information obtained from previous query computations with that in the query trace list of the current query computation.

3.7.1. Query Traffic Formats for Algorithm IV

Queries have the form

Q (Sender, Receiver, U, T, QTL)

Each field in a *query* has the same meaning as in Algorithm III. In the field QTL, each value M_{i_k} shows the number of messages accepted by process P_{i_k}.

Replies have the form

R (Sender, Receiver, T, QTL)

Each field in a *reply* has the same meaning as in Algorithm III.

Cancels have the form

C (Sender, Receiver, QTL)

Each field in a *cancel* has the same meaning as in Algorithm III.

Informs have the form

I (Sender, Receiver, M, T)

Each field in an *inform* has the same meaning as in Algorithm III.

3.7.2. Local Variables for Algorithm IV

Each process maintains twelve arrays of local variables and four scalar variables. One of the arrays is two-dimensional, and the other eleven arrays are one-dimensional of length N, where N is the number of processes in the system. The twelve arrays and four variables, as maintained by process P_k when processing a query computation initiated by

P_i, are described below.

LATEST(N,2N)

This array is the same as in Algorithm III.

ENGAGER(N)

This array is the same as in Algorithm III.

PRIOR(N)

This array is the same as in Algorithm III.

STARVE(N)

This array is the same as in Algorithm III.

TIMES(N)

This array is the same as in Algorithm III.

TARGET(N)

This array is the same as in Algorithm III.

DEPENDENT(N)

This array is the same as in Algorithm III.

REACHABLES(N)

This array is the same as in Algorithm III.

EDGETYPE(N)

This array is the same as in Algorithm III.

LASTREP(N)

This array is the same as in Algorithm III.

RECENT(N)

The entry $RECENT(i)$ shows the most recent sequence number for process P_i that is known to P_k. P_k updates this entry with any available information such as a message received from P_i or any query traffic which has P_i, M_i in the query trace list. This entry does not necessarily reflect the most recent sequence number of P_i, although the entry does show a lower bound.

VERIFIED(N)

The entry $VERIFIED(i)$ shows the most recent sequence number for process P_i which P_k discovered by receiving either a *query* or *reply* directly from P_i. Any message sent from P_i to P_k after the *query* or *reply* must necessarily have been sent after P_i had increased its sequence number, since messages are only sent by executing processes, *queries* and *replies* are sent only by idle processes, and we assumed sequenced message delivery.

LASTTIME

This variable is the same as in Algorithm III.

STATE

This variable is the same as in Algorithm III.

DEADLK

This variable is the same as in Algorithm III.

URGENCY

This variable is the same as in Algorithm III.

In the description of the events for Algorithm IV (shown below), and in the code (listed in Appendix D), the one-dimensional array QTL is used just as it was for Algorithms II and III.

3.7.3. Events for Algorithm IV

For Algorithm IV, the following events are of interest:

(1) An idle process receives a message.

(2) A process receives a *query*.

(3) A process receives a *reply*.

(4) A process receives a *cancel*.

(5) A process receives an *inform*.

(6) A process sends a message.

(7) An executing process changes state to idle.

(8) A process becomes idle for time T_1 since last executing or last receiving an *inform*. This is when the process initiates its own query computation.

(9) A process becomes idle for time T_2 (where $T_2 \leq T_1$) since last executing or last receiving an *inform*. This is when *queries* being held are restarted.

The actions taken by a process when each of these events occurs are described below. It is assumed, but not shown, that all query traffic received is first checked for validity. Events in Algorithm IV which are the same as those in Algorithm III are noted as such. The variable N contains the number of processes in the system. The pseudo code for

Algorithm IV is listed in Appendix D.

(1) When an idle process P_r receives a message M($P_{k,}$ $P_{r,}$ $M_{k,}$ $\cdots$) from P_k and begins executing, it takes the following actions:

(a) Increases its own sequence number by 1. The sequence number is stored in $RECENT(r)$.

(b) Changes $RECENT(k)$ to reflect the sequence number M_k.

(c) Performs the same actions as required for Algorithms II and III.

(2) When a process P_r receives a ***query*** Q ($P_{k,}$ $P_{r,}$ U, T, QTL), it takes the following actions:

(a) For each pair $P_{l,}$ M_l in the QTL, updates the entry $RECENT(l)$ to be M_l if M_l is greater than $RECENT(l)$.

(b) Updates $VERIFIED(k)$ to be M_k. (The pair $P_{k,}$ M_k must be in the QTL).

(c) If P_r is executing and has seen the query computation before (P_r is in the QTL), then

- Discards the ***query*** just received.

(d) If P_r is executing and has not seen the query computation before, then

- Adds itself to the end of the query trace list using its current sequence number.
- Updates the time field T in the ***query***.
- Stores a copy of the ***query***.
- Sends an ***inform*** to the query initiator.

(e) If P_r is not executing, then checks for any held query computations for which a ***reply*** can now be sent to the engager.

(f) If P_r is deadlocked, then

- If P_r is not in the QTL, adds P_r and all members of its reachable set which are not already in the QTL to the QTL. If process P_l is added from the reachable set, then sequence number $RECENT(l)$ is used. (In this case, $RECENT(l)$ must be the same as $VERIFIED(l)$ so either could be used).
- Sends a ***reply*** to process P_k.

(g) If P_r is idle, and P_r is in the QTL, then

- If P_r has been continuously idle since receiving the query computation the first time ($RECENT(i) = M_i$), then sends a ***reply*** to P_k. Otherwise discards the ***query*** during the validity check.

(h) If P_r is idle and P_r is not in the QTL, then

- adds itself (with sequence number $RECENT(r)$) to the QTL.
- If every process P_l in the dependent set is either the engager or appears in the QTL with sequence number M_l which is equal to $VERIFIED(l)$, then sends a ***reply*** back to process P_k.
- Otherwise, there is a process P_l in the dependent set which is not the engager and which either has not seen the query computation before (P_l is not in the QTL) or which has sequence number M_l which is greater than $VERIFIED(l)$.
    - Stores the ***query***.
    - Sets $STARVE(i)$ to 0.
    - If P_r has been idle for T_2 time since last receiving an ***inform*** or executing, and if the $URGENCY$ of the ***query*** just received is at least as high as the highest priority of any ***query*** currently received, sends a ***query*** to the first process P_l in the dependent set which is not the engager and which does not appear in the QTL with sequence number M_l equal to $VERIFIED(l)$. The ***query*** being sent creates either a tree edge (if P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). Sets $EDGETYPE(i)$ accordingly.

(3) When a process P_r receives a ***reply*** R (P_k, P_r, T, QTL), it takes the following actions:

(a) For each pair P_l, M_l in the QTL, updates the entry $RECENT(l)$ to be M_l if M_l is greater than $RECENT(l)$.

(b) Updates $VERIFIED(k)$ to be M_k. (The pair P_k, M_k must be in the QTL).

(c) Updates $LASTREP(i)$ and $LATEST(i,*)$.

(d) If every process P_l in the dependent set appears in the QTL with sequence number M_l equal to $VERIFIED(l)$, and if P_r is the query initiator, then

- Declares DEADLOCK for P_r and all processes in the QTL.

  For each process P_j appearing in the QTL, sets $REACHABLES(j)$ to TRUE.

- For each *query* which is either being held, or which is outstanding but did not create a tree edge,
  - Shortens the query trace list so that P_r is the last process in the query trace list.
  - For each such *query*, augments the query trace list to include all processes in the reachable set of P_r. Each process P_l added from the reachable set uses sequence number $VERIFIED(l)$.
  - For each such *query*, sends a *reply* to the engager process.

(e) If every process P_l in the dependent set (except possibly the engager) appears in the QTL with sequence number M_l equal to $VERIFIED(l)$, and P_r is not the query initiator, then

- If the query computation has the highest priority of any received, increases $STARVE(l)$ by one for each l where P_l is the initiator of a query computation currently being held by P_r.
- Sends a *reply* to the engager process.

(f) If P_r is not executing, then checks for any held query computations for which a *reply* can now be sent to the engager.

(g) If P_r has been idle T_2 time since last executing or receiving an *inform*, then

- Determines the highest priority of any *query* received. Selects all *queries* it is has received which have an $URGENCY$ at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. A process P_l in the dependent set which appears in the QTL with sequence number M_l equal to $VERIFIED(l)$ does not need to be queried. Any *query* being sent creates either a tree edge (if the target process P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). For each *query* sent, sets $EDGETYPE(i)$ accordingly.

(4) When a process P_r receives a *cancel* C (P_k, P_r, QTL), it takes the following actions:

(a) If P_r is executing and the query computation associated with the *cancel* has the highest priority of any received, increases $STARVE(l)$ by one for each l where P_l is the initiator of a query computation currently being held by P_r.

(b) Processes the *cancel* just as for Algorithms II and III.

(c) For each of the highest urgency *queries* remaining, if P_k has been idle for time T_2 since last executing or receiving an *inform*, then ensures that the *query* is not being held, since the *cancel* could have stopped the highest urgency query computation.

- Determines the highest priority of any *query* received.
- Selects all *queries* it is has received which have an *URGENCY* at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, it must be sent to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. A process P_l in the dependent set which appears in the *QTL* with sequence number M_l equal to $VERIFIED(l)$ does not need to be queried. $LASTREP(i)$ shows the progression through the dependent set. Any *queries* being sent create either tree edges (if the target processes P_l are not in the *QTL*) or different types of edges (forward, back, or cross edges). For each *query* sent, sets $EDGETYPE(i)$ accordingly.

(5) When a process P_r receives an *inform* I (P_k, P_r, T), the actions required are the same as for Algorithms II and III.

(6) When a process P_k sends a message, no query computation action is required.

(7) When an executing process P_k changes state to idle, it takes the following actions:

(a) Changes its *STATE* flag to show it is idle.

(b) Sets the lasttime that it knew that it was not deadlocked.

(c) For each *query* currently being held by P_k, changes the sequence number associated with P_k in the *QTL* to be $RECENT(k)$. These *queries* will be held until P_k becomes idle for T_2 time since last executing or receiving an *inform*.

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle or since receiving the last *inform*, it takes the following actions:

(a) Initiates a new query computation by building a *query* and storing it.

(b) If the *URGENCY* of the *query* just created is as high or higher than any received *queries*, then sends the new *query* to the first process in its dependent set. This creates a tree edge. Sets *EDGETYPE* accordingly.

(9) When a process P_i becomes idle for time T_2 (where $T_2 \geq T_1$) since changing state from executing to idle or since receiving the last *inform*, it takes the following actions:

(a) Determines the highest priority of any *query* received.

(b) Selects all *queries* it is has received which have an $URGENCY$ at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. A process P_l in the dependent set which appears in the QTL with sequence number M_l equal to $VERIFIED(l)$ does not need to be queried. $LASTREP(i)$ shows the progression through the dependent set. Any *queries* being sent create either tree edges (if the target processes P_l are not in the QTL) or different types of edges (forward, back, or cross edges). For each *query* sent, sets $EDGETYPE(i)$ accordingly. Lower urgency *queries* are held.

3.8. Algorithm V

As stated in Section 3.2, one of our goals of a deadlock detection scheme is to make the resolution of deadlock as efficient as possible. In any directed graph, there are one or more strongly connected components [AhHo74, page 189]. If process A is deadlocked, then the entire reachable set of A is also deadlocked. To break all deadlocks in the reachable set of A, we need to find all strongly connected components which have the property that the dependent set of any process in any component is also contained in the component. Each such strongly connected component forms what we term a ***minimal deadlocked set***, and deadlock can only be broken by having actions taken by or applied to at least one of the processes in the minimal deadlocked set [Leun83]. Therefore, an efficient deadlock resolution algorithm should operate only on minimal deadlocked sets. Information on the identities of processes in the minimal deadlocked set will be needed to implement such algorithms.

In the example of Figure 3, processes B and C form a strongly connected component, as do processes E, F, and G. Suppose that process A, upon becoming idle, first sends an engaging *query* to B and the following conditions hold:

- *A* is not in the reachable set of *B*,
- *B* is deadlocked, and
- No member of the reachable set of *B* sees the query computation before *B*.

We want to argue that if *B* replies to its engager, *B* can discover it is deadlocked and know the identities of all members of its deadlocked set without having to initiate its own query computation. Once the *query* reaches *B*, *B* adds itself to the query trace list. Since no member of the reachable set of *B* has seen the query computation before *B*, it must happen that as long as *B* has the query computation outstanding, all *queries* and *replies* for this query computation occur between members of the reachable set of *B*, and all these processes are added to the query trace list after *B*. Therefore, if *B* can learn at the time it replies to its engager that this situation has occurred, it can declare deadlock for itself and all succeeding processes in the query trace list. For *B* to learn this, the query trace list must have a boolean flag for each process saying whether or not that process has a member of its reachable set which preceeds it in the query trace list.

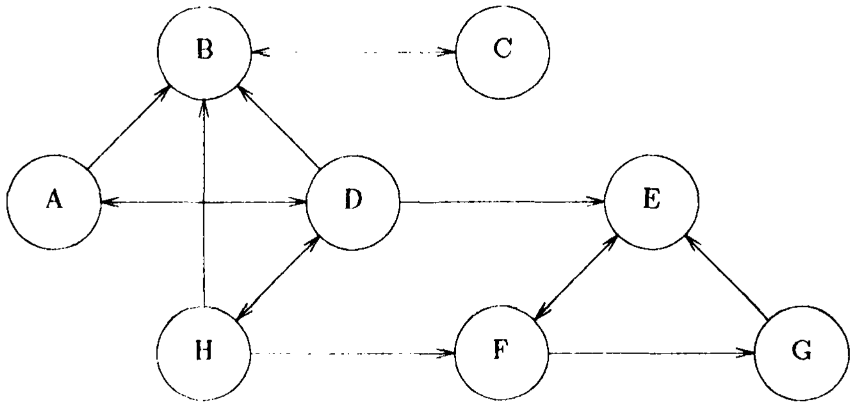


Figure 3
Minimal Deadlocked Set Example

By some careful manipulation of these boolean flags, it is also possible to discover the minimal deadlocked sets as the query computation progresses. Recall that a minimal deadlocked set is a deadlocked set which is also a strongly connected component. Therefore, the reachable set of any process in a minimal deadlocked set is also in the minimal deadlocked set. Also, if some process A in the minimal deadlocked set receives an engaging *query* from a process B not in the minimal deadlocked set, it will not reply to its engager until all processes in its reachable set have seen the query computation. All these processes must see the query computation after process A, and therefore will appear in the query trace list after A. Moreover, each of these processes, when querying their dependent sets, will not query a process outside the minimal deadlocked set. Therefore, if A, after querying its dependent set, can determine that

– No process in the reachable set of A occurs before A in the query trace list, and

– There is no minimal deadlocked set in its reachable set which does not include A,

then A can say it is a member of a minimal deadlocked set, and that all processes following it in the query trace list are also members of the minimal deadlocked set. This also says, of course, that B, which is the engager of A, cannot be in any minimal deadlocked set. Process A can then notify other members of the minimal deadlocked set if it wishes, and a deadlock resolution algorithm may be invoked with the identities of all processes in the minimal deadlocked set as input.

3.8.1. Query Traffic Formats for Algorithm V

Queries have the form

Q (Sender, Receiver, U, T, QTL)

In each *query*,

(1) Each field except QTL has the same meaning as in Algorithm IV.

(2) QTL is a list of process/boolean flag/sequence number triples $P_{i_1}, F_{i_1}, M_{i_1}, P_{i_2}, F_{i_2}, M_{i_2}, \cdots, P_{i_j}, F_{i_j}, M_{i_j}$ and is referred to as the *query trace list*. A process P_{i_k} which appears in this sequence has been queried during the current query computation and is assumed to be idle by the sender. F_{i_k} is a boolean flag used for finding minimal deadlocked sets. M_{i_k} is the sequence number associated with process P_k.

Replies have the form

R (Sender, Receiver, T, QTL)

Each field in a *reply* has the same meaning as in a *query*.

Cancels have the form

C (Sender, Receiver, QTL)

Each field in a *cancel* has the same meaning as in Algorithm IV. The boolean flags are not included in the QTL for a *cancel*.

Informs have the form

I (Sender, Receiver, M, T)

Each field in an *inform* has the same meaning as in Algorithm IV.

3.8.2. Local Variables for Algorithm V

Each process maintains eleven arrays of local variables and five scalar variables. One of the arrays is two-dimensional, and the other ten arrays are one-dimensional of length N, where N is the number of processes in the system. The eleven arrays and five variables, as maintained by process P_k when processing a query computation initiated by process P_i, are described below.

LATEST(N,2N)

This array is the same as in Algorithm IV except that the boolean flag F_{i_k} in the query trace list is stored as the sign field of the associated sequence number M_{i_k}. The array *REACHABLES* as used in Algorithm IV is insufficient for remembering minimal deadlocked sets. Therefore, if a process P_i discovers deadlock, it will store the applicable portion of the query trace list in the array $LATEST(i,*)$ whether or not it initiated the query computation. Once the flag *DEADLK* is set to true, $LATEST(i,*)$ will not be changed. Note that if a process P_i declares deadlock based on a query computation initiated by another process, then the reachable set of P_i will consist of all processes in the query trace list between P_i and the end of the query trace list, inclusive, at the time P_i replies to its engager.

ENGAGER

This array is the same as in Algorithm IV.

PRIOR

This array is the same as in Algorithm IV.

STARVE(N)

This array is the same as in Algorithm IV.

TIMES(N)

This array is the same as in Algorithm IV.

TARGET(N)

This array is the same as in Algorithm IV.

DEPENDENT(N)

This array is the same as in Algorithm IV.

EDGETYPE(N)

This array is the same as in Algorithm IV.

LASTREP(N)

This array is the same as in Algorithm IV.

RECENT(N)

This array is the same as in Algorithm IV.

VERIFIED(N)

This array is the same as in Algorithm IV.

LASTTIME

This variable is the same as in Algorithm IV.

STATE

This variable is the same as in Algorithm IV.

DEADLK

This variable is the same as in Algorithm IV.

URGENCY

This variable is the same as in Algorithm IV.

In the description of the events for Algorithm V (shown below), and in the pseudo code (listed in Appendix F), the one-dimensional array QTL is used just as it was for Algorithm IV except that $QTL(2k)$ now holds both the boolean flag F_{i_k} and the sequence number M_{i_k}.

3.8.3. Events for Algorithm V

For Algorithm V, the following events are of interest:

(1) An idle process receives a message.

(2) A process receives a *query*.

(3) A process receives a *reply*.

(4) A process receives a *cancel*.

(5) A process receives an *inform*.

(6) A process sends a message.

(7) An executing process changes state to idle.

(8) A process becomes idle for time T_1 since last executing. This is when the process initiates its own query computation.

(9) A process becomes idle for time T_2 (where $T_2 \geq T_1$) since last executing. This is when *queries* being held are restarted.

The actions taken by a process when each of these events occurs are described below. It is assumed, but not shown, that all query traffic received is first checked for validity. Events in Algorithm V which are the same as those in Algorithm IV are noted as such. The variable N contains the number of processes in the system. The pseudo code for Algorithm V is listed in Appendix F.

(1) When an idle process P_r receives a message M ($P_k, P_r, M_k, \cdots$) from P_k and begins executing, the actions required are the same as for Algorithm IV.

(2) When a process P_r receives a *query* Q (P_k, P_r, U, T, QTL), it takes the following actions:

(a) If P_r has a *query* outstanding for an earlier query computation for the same initiator, and the *query* it sent out earlier created a tree edge, then sends a *cancel* to stop the earlier query computation.

(b) For each triple P_l, F_l, M_l in the QTL, updates the entry $RECENT(l)$ to be M_l if M_l is greater than $RECENT(l)$.

(c) Updates $VERIFIED(k)$ to be M_k. (The triple P_k, F_k, M_k must be in the QTL).

(d) If P_r is executing and has seen the query computation before (P_r is in the QTL), then discards the *query* just received.

(e) If P_r is executing and has not seen the query computation before, then

Adds itself to the end of the query trace list using its current sequence number. Sets its boolean flag F_r to FALSE ( ).

Updates the time field T in the *query*.

Stores a copy of the *query*.

Sends an *inform* to the query initiator.

(f) If P_r is not executing, then checks for any held query computations for which a *reply* can now be sent to the engager.

(g) If P_r is deadlocked and P_r is a member of a minimal deadlocked set, then

- If P_r already has an outstanding *query* for this query computation, then
  - Adds P_r and all members of its reachable set which are not already in the QTL to the QTL using the order in which they appear in $LATEST(r,*)$.
  - Finds the first process P_s in the QTL which is in the reachable set of P_r. Does not change the value of flag F_s. For all other flags F_l in the QTL, if P_l is in the reachable set of P_r, then sets flag F_l to be TRUE (+).
- If P_r does not already have an outstanding *query* for this query computation and the engaging process P_k is in the reachable set of P_r, then
  - Adds P_r and all members of its reachable set which are not already in the QTL to the QTL using the order in which they appear in $LATEST(r,*)$. The boolean flags are used just as they appear in $LATEST(r,*)$.
  - Finds the first process P_s in the QTL which is in the reachable set of P_r. Does not change the value of flag F_s. For all other flags F_l in the QTL, if P_l is in the reachable set of P_r, then sets flag F_l to be TRUE (+).
- If P_r does not already have an outstanding *query* for this query computation and the engaging process P_k is not in the reachable set of P_r, then
  - Sets the flag F_k to TRUE (+).
  - Adds P_r and all members of its reachable set which are not already in the QTL to the QTL using the order in which they appear in $LATEST(r,*)$. The boolean flags are used just as they appear in $LATEST(r,*)$.
- Sends a *reply* to process P_k.

(i) If P_r knows it is deadlocked and P_r is not a member of a minimal deadlocked set, then

- Sets the flag F_k is the QTL to TRUE (+).
- Adds P_r and all members of its reachable set which are not already in the QTL to the QTL using the order in which they appear in $LATEST(r,*)$. The boolean flags are used just as they appear in $LATEST(r,*)$.
- Sends a *reply* to process P_k.

(j) If P_r is idle, and P_r is in the QTL, then

- If P_r has been continuously idle since receiving the query computation the first time $(RECENT(i) = M_i)$, then sends a *reply* to P_k. Otherwise discards the *query* during the validity check.

(k) If P_r is idle and P_r is not in the QTL, then

- Adds itself (with sequence number $RECENT(r)$ and boolean flag F_r = FALSE (-)) to the QTL.
- If a process in the dependent set of P_r is already in the QTL, then
    - Finds the first process P_l in the QTL which is in the dependent set of P_r. Sets all boolean flags between F_l and F_r (non-inclusive) to TRUE (+).
    - Sets the boolean flag F_r to TRUE (+).
- If every process P_l in the dependent set is either the engager or appears in the QTL with sequence number M_l which is equal to $VERIFIED(l)$, then
    - If flag F_r is still FALSE (-), then declares DEADLOCK for P_r and all processes occurring after P_r in the QTL. Changes the QTL so that all flags ahead of F_r are set to TRUE (+) and all flags following F_r have the pattern (------+).
    - Sends a *reply* back to process P_k.
- Otherwise, there is a process P_l in the dependent set which is not the engager and which either has not seen the query computation before (P_l is not in the QTL) or which has sequence number M_l which is greater than $VERIFIED(l)$.
    - Stores the *query*.
    - Sets $STARVE(i)$ to 0.

(k) If P_r has been idle for T_2 time since last receiving an *inform* or executing, and if the $URGENCY$ of the *query* just received is at least as high as the highest priority of any *query* currently received, sends a *query* to the first process P_l in the dependent set which is not the engager and which does not appear in the QTL with sequence number M_l equal to $VERIFIED(l)$. The *query* being sent creates either a tree edge (if P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). Sets $EDGETYPE(i)$ accordingly.

(3) When a process P_r receives a *reply* R (P_k, P_r, T, QTL), it takes the following

actions:

(a) For each triple P_l, F_l, M_l in the QTL, updates the entry $RECENT(l)$ to be M_l if M_l is greater than $RECENT(l)$.

(b) Updates $VERIFIED(k)$ to be M_k. (The triple P_k, F_k, M_k must be in the QTL).

(c) Updates $LASTREP(i)$ to be P_k.

(d) Adds to $LATEST(i,*)$ any triple P_l, F_l, M_l in the QTL, provided P_l is not in $LATEST(i,*)$.

(e) For any triple P_l, F_l, M_l in the QTL which preceeds P_r, updates the flag F_l in $LATEST(i,*)$ to reflect the value in the QTL.

(f) If every process P_l in the dependent set of P_r (except possibly the engager of P_r) appears in the QTL with sequence number M_l equal to $VERIFIED(l)$, and if P_r is the query initiator, then

- Declares DEADLOCK for P_r and all processes in the QTL. Sets $DEADLK$ to TRUE.
- If F_r is FALSE, P_r knows it is a member of a minimal deadlocked set. Changes F_r and all flags following it in the QTL except the last one to be FALSE. The minimal deadlocked set now has a flag pattern of (––––––+). Updates $LATEST(r,*)$ to reflect the values in the QTL.
- For each *query* which is either being held, or which is outstanding but did not create a tree edge,
    - Augments the QTL to include all processes in the reachable set of P_r. Each process P_l added from the reachable set uses sequence number $VERIFIED(l)$. These processes in the reachable set of P_r are found in $LATEST(r,*)$ and must be added so as to preserve the identity of all minimal deadlocked sets. Processes not already in the QTL are simply added in the order in which they appear in $LATEST(r,*)$. This ensures that each minimal deadlocked set discovered by P_r is identified by the special pattern of boolean flags (–––––+).
    - If P_r is not a member of a minimal deadlocked set, then changes the flag of the engager process to be TRUE (+).
    - If P_r is a member of a minimal deadlocked set, and the engager of P_r is not a member of the reachable set of P_r, then changes the flag of the engager process to be TRUE (+).
    - If P_r is a member of a minimal deadlocked set, and the engager of P_r is in the reachable set of P_r (i.e. also in the same minimal deadlocked set), then finds the first process P_s in the QTL which is in the reachable set of P_r. Does not change the value of flag F_s. For all other flags F_l in the QTL, if P_l is in the reachable set of P_r, then sets flag F_l to be TRUE (+). Thus the minimal deadlocked set will have

the flag pattern (–+++++) until the reply reaches the first member of the QTL which is a member of the minimal deadlocked set. This process will change the pattern to (–––––+).

– For each such ***query***, sends a ***reply*** to the engager process.

(g) If every process P_l in the dependent set (except possibly the engager) appears in the QTL with sequence number M_l equal to $VERIFIED(l)$, and P_r is not the query initiator, then

- If the priority of the ***reply*** just received is at least as high as the highest priority query computation received, then for each query computation being held, increases its $STARVE$ count by 1.

- If F_r is FALSE (–), then

  - The engager cannot be a member of the dependent set of P_r. Declares DEADLOCK for P_r and all processes in the QTL which follow P_r. Stores the rear portion of the QTL beginning with P_r in $LATEST(r,*)$. Changes the flags in $LATEST(r,*)$ and in the portion of the QTL starting with F_r to be the special pattern (–––––+). These processes form a minimal deadlocked set. Sets $DEADLK$ to TRUE. Sends a ***reply*** to the engager.

  - For each ***query*** which is either being held or which is outstanding but did not create a tree edge,

    - For each such ***query***, augments the query trace list to include all processes in the reachable set of P_r. Each process P_l added from the reachable set uses sequence number $VERIFIED(l)$. These processes in the reachable set of P_r are found in $LATEST(r,*)$ and must be added so as to preserve the identity of all minimal deadlocked sets. Processes not already in the QTL are simply added in the order in which they appear in $LATEST(r,*)$.

    - If the engager of P_r is not a member of the reachable set of P_r, then changes the flag of the engager process to be TRUE (+). The minimal deadlocked set to which P_r belongs will be at the end of the QTL with the special pattern of boolean flags (–––––+).

    If the engager of P_r is in the reachable set of P_r (ie also in the same minimal deadlocked set), then finds the first process P_s in the QTL which is in the reachable set of P_r. Does not change the value of flag F_s (it will be FALSE). For all other flags F_l in the QTL, if P_l is in the reachable set of P_r, then sets flag F_l to be TRUE (+). Thus the minimal deadlocked set will have the flag pattern (–+++++) until P_s receives a ***reply***. P_s will change the pattern to (–––––+).

- For each such *query*, sends a *reply* to the engager process.

(h) If P_r has been idle T_2 time since last executing, then

- Determines the highest priority of any *query* received. Selects all *queries* it is has received which have an *URGENCY* at least this high. For each such *query* (assume P_i is the initiator), if the *query* is being held, sends it to the next process in the dependent set which needs to be queried and sets $STARVE(i)$ to 0. $LASTREP(i)$ shows the progression through the dependent set. A process P_l in the dependent set which appears in the QTL with sequence number M_l equal to $VERIFIED(l)$ does not need to be queried. Any *query* being sent creates either a tree edge (if the target process P_l is not in the QTL) or a different type of edge (forward, back, or cross edge). For each *query* sent, sets $EDGETYPE(i)$ accordingly.

(4) When a process P_r receives a *cancel* C $(P_{k,} P_{r,} QTL)$, the action required is the same as for Algorithm IV.

(5) When a process P_r receives an *inform* I $(P_{k,} P_{r,} T)$, actions required are the same as for Algorithm IV.

(6) When a process P_k sends a message, no query computation action is required.

(7) When an executing process P_k changes state to idle, the actions required are the same as for Algorithm IV.

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, the actions required are the same as for Algorithm IV. The flag F_i in the QTL is set to FALSE.

(9) When a process P_i becomes idle for time T_2 (where $T_2 < T_1$) since changing state from executing to idle, the actions required are the same as for Algorithm IV.

CHAPTER 4.

Proofs of Correctness

4.1. Introduction

This chapter presents the proofs of the algorithms described in Chapter 3. Theorems 1 and 3 state that all true deadlocks are detected. Theorem 2 says that there are no false detections. Theorems 4, 5, and 6 apply only to Algorithm V; they show that a query computation identifies all minimal deadlocked sets and that at least one process from each minimal deadlocked set discovers deadlock for itself during the course of the query computation. Furthermore, when such a process discovers deadlock, it knows the identities of the other processes in the minimal deadlocked set. Definitions used in these theorems and their proofs are shown in Appendix F. Throughout this chapter, we make the following assumptions:

(1) There are only a finite number of processes in the system,

(2) There is a finite number of priority classes for query computations,

(3) No process is infinitely lazy, and

(4) Messages (and query computations) are delivered in finite amounts of time and in the order sent.

4.2. Theorems and Proofs

Theorem 1, in conjunction with theorem 3, shows that all true deadlocks are detected.

Theorem 1: If the initiator of a query computation is deadlocked when it initiates the computation, it will eventually declare itself "deadlocked".

Proof: For Algorithms I and II: Consider a set S of N processes, including the initiator P, which is deadlocked when P initiates a query computation. Processes in S cannot change their dependent sets, and there are no messages in transit between processes in S. Hence, after P initiates the query computation, no process in S can become executing. When receiving a query belonging to the query computation for the first time (an engaging query), each process adds itself to the query trace list and queries each process in its dependent set once in turn. Each process, upon receiving a query, either sends the corresponding reply or sends another query. Since each process can have at most $N-1$ processes in its dependent set, there can be at most $N(N-1)$ queries belonging to the query computation initiated by P. A reply is sent only when the corresponding query is received for the first time, and for any query received, exactly one reply has been sent. Hence, there can be at most $N(N-1)$ replies for the query computation. Therefore, after some finite amount of time, no more queries will be sent, and the corresponding reply for each query has been sent and received. By this time, the initiator has queried all members of its dependent set, has received the corresponding replies from all of them, and hence, declares itself deadlocked.

For Algorithms III, IV, and V: The above argument holds if no process in the system has a higher priority than P, since the highest priority query computations are never suspended by other query computations. It also holds when all processes with priorities higher than P are in the set S. In this case, each of the highest priority query computations initiated by processes in S will finish. Once they do, each of the next highest priority query computations becomes the highest priority query computation. Each of these also finishes, until finally the query computation initiated by P finishes, and P detects deadlock.

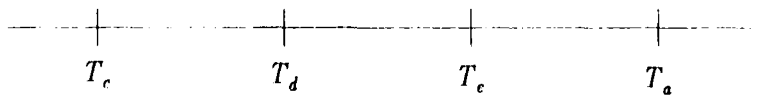
Suppose, however, that there are exactly M processes not in S which have priorities higher than P and each of which has a reachable set containing P. Let H_{max} be the highest priority which may be assigned to a query computation. We will show that the query computation initiated by P cannot be suspended indefinitely by query computations initiated by these M processes, causing P to fail in detecting deadlock. To show that this cannot occur, we show that the highest priority query computation QC_0 initiated by any process in S will make progress. More specifically, if the query computation QC_0 is suspended by a higher priority query computation initiated by one of the M processes not in S, QC_0 will eventually increase its urgency by 1. As defined in Chapter 3, urgency is the sum of priority of the query computation and the integer quotient of the starvation count divided by a user-defined finite number C. The starvation count of a suspended query computation is increased by 1 each time the process which suspended the query computation sends a reply or receives a cancel.

We now show that QC_0 will raise its starvation count by 1 in finite time. Either QC_0 is making progress, in which case the proof is completed, or else it is suspended by an engaging query from a higher priority query computation (say QC_1) received by some process Q in S from a process not in S. Either QC_1 is making progress, or else it is suspended by a higher priority query computation (say QC_2) initiated by another process not in S. Since there are M processes not in S which have a priorities higher than the highest priority process in S, there is a query computation QC_M with the highest priority. Note that the priority of QC_M is less than or equal to $H_{\max}$. QC_M, and all other query computations with this priority, make progress among processes in S. Process Q in S which receives an engaging query from any of these highest priority query computations will either eventually reply to its engager, or else receive a cancel for the query computation. Starvation counts of all query computations suspended by Q will be increased by 1. Hence, they make progress. That is, they will eventually finish (replies are sent to engagers) or else they will be terminated with a cancel. In either case, all processes which receive engaging queries belonging to QC_{M-1} will either send a reply or receive a cancel for QC_{M-1}. All query computations suspended by these processes thus increase their starvation counts by 1, and hence make progress. Because the number of priority classes is finite, QC_0 will eventually increase its starvation count by 1, and hence make progress. By continuing this argument, we can conclude that the starvation count of QC_0 will eventually reach the level C, causing its urgency to increase by 1. Similarly, the urgency of QC_0 eventually will reach a level sufficient to ensure its progress. In particular, since the priority of any query computation is bounded by $H_{\max}$, QC_0 begins to make progress whenever its urgency is as high or higher than the priority of the query computation which caused process Q to suspend QC_0.

Theorem 2 states that a process does not detect a false deadlock. It will be proven by contradiction using Lemma 1.

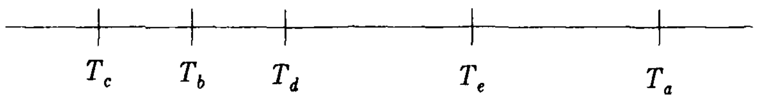
Lemma 1: Suppose that a process P_k sends a reply to its engager and subsequently becomes executing at some time T_a during a query computation. Then there exists a process P_r in the dependent set of P_k which receives a query for that query computation and subsequently becomes executing at some time $T_b < T_a$.

Proof: Suppose that a process P_e sends process P_k an engaging query at some time T_c. Thus, P_e is the engager of P_k. Suppose that P_k receives this query at some time $T_d > T_c$ and sends a reply to P_e at some time $T_e > T_d$. P_k was idle. It can only become executing if it receives a message from some process P_r in its dependent set. Suppose that P_k receives a message from P_r and becomes executing at some time $T_a > T_e$. The relationships between T_a, T_c, T_d, and T_e are as shown below. There are two cases: either P_r is P_e, or P_r is not P_e.



Case 1: Suppose that P_r is P_e.

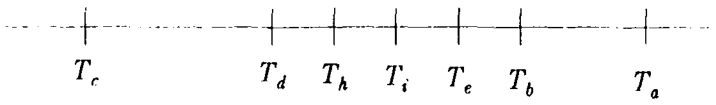
Since message delays are finite, P_r sent the message at some time $T_b < T_a$. Also, since sequenced message delivery is assumed, the query was sent by P_r to P_k before the message. Hence, $T_c < T_b$ as shown below.



Queries are sent only by idle processes, and messages are sent only by executing processes. Hence P_r was idle at time T_c, received the query (or initiated it), and subsequently became executing at time $T_b < T_a$.

Case 2: Suppose that P_r is not P_e.

For Algorithms I, II, and III, P_k does not send a reply to P_r until it has queried and received replies from every process in its dependent set (except possibly the engager). Therefore, P_r received a query from P_k at some time $T_h > T_d$ and sent a reply to P_k at some time T_i where $T_h < T_i < T_e$ as shown below. Since P_k received a message from P_r after receiving a reply from P_r, the message from P_r was sent at some time T_b where $T_i < T_b < T_a$ as shown.



Hence, P_r was idle, received the query (or initiated it), and subsequently was executing at time $T_b < T_a$.

For Algorithms IV and V, P_k does not send a reply to P_e unless every process in its dependent set appears in the query trace list and the sequence number N_i matches the most recent sequence number M_i of some query or reply sent from P_i to P_k for every process id/sequence number pair $P_i\, N_i$ appearing in the query trace list. Therefore, when P_k sends a reply to P_e at time T_e, process P_r and its associated sequence number N_r are in the query trace list. P_k knows from some query or reply it received from P_r that any message subsequently received from P_r carries a sequence number greater than N_r. Therefore, the message received by P_k at time T_a carries a sequence number greater than N_r. Since P_r inserted itself with sequence number N_r in the query trace list, this implies that P_r became executing at some time $T_b < T_a$ after receiving the query computation.

Theorem 2 states that a process does not detect a false deadlock, and that whenever it detects a deadlock, it knows all the processes in its reachable set.

Theorem 2: If the initiator P of a query computation declares itself "deadlocked", then it belongs to a deadlocked set $\hat{S}$ containing itself and all the processes in its reachable set. Moreover, all the processes in the reachable set of P are known to P when deadlock is declared.

Proof: Let S be the set of processes which receive queries during the query computation. As described in Chapter 3, a process Q in S adds itself to the query trace list when it receives an engaging query. Every process replies to its engager only after it receives a reply for every query that it sent. After the initiat[illegible]ceives a reply for every query that it sent, it declares itself deadlocked. Therefore, when the initiator P declares itself deadlocked, every process in S has replied to its engager in the query computation.

We now show by contradiction that no process in S can become executing after sending a reply to its engager. In other words, processes in S are indeed deadlocked when P declares itself deadlocked. Suppose some processes in S become executing after replying to their engagers. Let Q be the first such process. From Lemma 1, if process Q becomes executing at time t_Q after replying to its engager, then there is a process R in the dependent set of Q which becomes executing at time t_R after receiving a query, and $t_R < t_Q$. Q cannot be the engager of R, since Q being the the engager of R means that R replied to Q before Q replied to its engager. If this happens, then Q is not the first process to execute after replying to its engager as it is supposed to be. Two possibilities remain: either R is in the engager chain of Q, or it is not. If it is not, then R must have received an engaging query from some other process and replied to that process before Q received its engaging query. Again, Q is not the first process to execute after replying to its engager if this happens. The only other possibility is that R is in the engager chain of Q. Q being the first process to execute after replying to its engager means that R has not replied to its engager at time t_R. Therefore, R still has an outstanding query at time t_R. When R executes at time t_R, it cancels its outstanding query. For Algorithm I, the query computation is halted, and for Algorithms II, III, IV, and V process R increases its sequence number in the query trace list. Thus, the reply which Q sent no longer belongs to the current query computation. It follows that no process in S becomes executing after sending a reply to its engager.

For Algorithms III, IV, and V, when a process that has already declared deadlock receives an engaging query, it adds the members of its reachable set to the query trace list and sends a reply to its engager without querying its dependent set. This means that the set S of processes may be a proper subset of the reachable set of P. To complete the proof, we need to show closure over the dependent set relationships for the deadlocked set declared by P. To do this, we first suppose that P is the first process in S to declare deadlock. We can conclude that the dependent set of every process in S is also in S, since every process in the dependent set of some process Q in S sees the query computation before Q replies to its engager, In this case, S is a deadlocked set. Every process in the reachable set of P receives an engaging query and adds itself to the query trace list. We can further conclude that the initiator P knows the identity of every process in its reachable set upon receiving the last reply. That is, the set $\hat{S}$ is the set S.

Suppose that P is not the first process in S to declare deadlock. We prove by induction that the set $\hat{S}$ containing P and its reachable set is a deadlocked set, and that P knows the identities of all processes in its reachable set. Let V be the first process to declare deadlock. By the argument above, V and its reachable set is a deadlocked set, and V knows the identities if all processes in its reachable set. Consider the set of processes S' which is the union of S and the reachable sets of those processes in S which have already declared deadlock. Suppose that the reachable set of any process W in S that declared deadlock before P is a deadlocked set and that W knows the identities of all processes in its reachable set. We first show that S' is also a deadlocked set. We have already shown that no member of S can become executing after sending a reply to its engager. None of the processes added to S to form S' can become executing either, since they are members of deadlocked sets. Therefore, S' is a deadlocked set. We now show that P learns the identities of all processes in its reachable set. Each process in S' is in the reachable set of P. Suppose there is a process U which is in the reachable set of P which is not in S'. There is a chain of processes $P, Q_1, Q_2, \ldots, U$ such that each process is in the dependent set of its predecessor. Each process either receives an engaging query from its predecessor, or else some ancestor in the chain has already declared deadlock. If no process in the chain has declared deadlock, then U receives a query, and hence U is a member of S. If some process in the chain has declared deadlock, then let Q_0 be the first one (in the chain order). Q_0 receives a query and sends a reply after including in the query trace list all members of its reachable set. Therefore, U will be in S'. This says every process in the reachable set of P is in S'. Therefore, the reachable set of P is S'. This also says that S' is $\hat{S}$. Because each process in S adds itself to the query trace list, and any process which already declared deadlock adds its reachable set to the query trace list, all processes in S' will appear in the query trace list when P declares deadlock. Thus, P knows the identities of all processes in its reachable set.

Theorem 3 is used in conjunction with theorem 1 to show that all true deadlocks are detected.

Theorem 3: (for Algorithm I)

If every process initiates a new query computation after it becomes idle or receives an inform, then every process in every deadlocked set will report "deadlocked".

Proof: Suppose a process P, upon becoming idle, initiates a query computation. Since all messages are delivered in finite time, P will eventually declare deadlock if the query computation is not halted. Hence, we need to consider only the case when the query computation initiated by P is halted. The query computation may be halted in one of three ways:

1) An idle process with an outstanding query receives a message and starts executing;

2) An executing process receives a query; or

3) An idle process receives a nonengaging query and the process has not been continuously idle since seeing the engaging query.

Consider cases 1 and 2. When some process Q halts a query computation, it sends an inform to P. Once the inform arrives, P will initiate a new query computation.

Now consider case 3. Let Q be the first process to halt the query computation. Suppose that Q receives the nonengaging query from a process W, where W is not necessarily different from P. If process Q is in the engager chain of W, then we have case 1. That is, P will receive an inform from Q. If Q is not in the engager chain of W when Q begins executing at some time T_Q, then Q received an engaging query, sent a reply to its engager, received a message and became executing, and then received the nonengaging query from W. By Lemma 1, some process R in the dependent set of Q received an engaging query and subsequently became executing at some time $T_R < T_Q$. Process R cannot have an outstanding query at time T_R, since it would have stopped the query computation when it began executing. Therefore, R replied to its engager before time T_R. This leads to a contradiction to the supposition that Q is the first process to stop the query computation.

Theorem 3: (for Algorithms II, III, IV, and V)

If every process initiates a new query computation whenever it becomes idle, then every process in every deadlocked set will report "deadlocked".

Proof: From theorem 1, the last process to become idle in a deadlocked set will detect deadlock. We now show that when all processes in a deadlocked set become idle, the query computations initiated by the processes other than the last process to become idle in the deadlocked set will run to completion. When they complete, their initiators will each declare deadlock. To show this, we note that query computations can be halted only by the initiator when it executes. When a process Q in the engager chain of a query computation executes, it suspends the query computation (and sends a *cancel* if needed). Then Q shortens the engager chain so that it is the last process on the engager chain. Once Q becomes idle, it resumes the query computation by querying its dependent set.

Theorems 4, 5, and 6 only apply to Algorithm V. These theorems establish that a deadlocked process will recognize all minimal deadlocked sets in its reachable set. Lemma 2 is used in the proof for theorem 5.

Lemma 2: If a process P replies to its engager, then all processes in the reachable set of P appear in the query trace list of the reply.

Proof: Suppose a process P replies to its engager, and Q is in the reachable set of P but not in the query trace list of the reply sent from P to its engager. There is a path from P to Q such that each process in the path is in the dependent set of its predecessor. Let R be the first process on this path which is not in the query trace list. We note that no process between P and R has declared deadlock, since if there were a process between P and R which had declared deadlock, it would have added itself and all members of its reachable set to the query trace list. That is, R would be in the query trace list. Let V be the predecessor of R in the path from P to Q. Since V is in the query trace list, it will have seen the query computation and added itself to the query trace list. In other words, V has queried each each member of its dependent set which is not already in the query trace list before replying to its engager. Therefore, V has queried R. This is a contradiction, since this implies R added itself to the query trace list before replying to V.

Theorem 4: If a process P in a minimal deadlocked set initiates a query computation, it will declare deadlock for itself and all members of its reachable set. Moreover, P recognizes that its reachable set is a minimal deadlocked set.

Proof: From theorem 1, the initiator P eventually declares deadlock. From theorem 2, P knows the identity of every process in its reachable set. We now show that P recognizes its reachable set as a minimal deadlocked set. Every process in the reachable set of P is in the same strongly connected component as P. P initializes its boolean flag F_P in the query trace list to FALSE. P recognizes its reachable set as a minimal deadlocked set if and only if the boolean flag F_P in the last reply received by P is FALSE. The only way in which the boolean flag F_P in the query trace list can be set to TRUE is if some process Q in the reachable set of P recognizes that Q is in a minimal deadlocked set, and that the engager of Q is not. To show that this cannot happen, we suppose some process Q in the reachable set of P recognizes that Q is in a minimal deadlocked set. When Q replies to its engager, the boolean flag F_Q is FALSE. All processes in the dependent set of Q appear after Q in the query trace list, since otherwise Q would have set F_Q to TRUE. Also, for a process R appearing after Q in the query trace list, all processes in the dependent set of R appear after Q in the query trace list. This says there is no directed path from Q to P. This is a contradiction, since P and Q are in the same strongly connected component. Therefore, our supposition is not true; P recognizes it is in a minimal deadlocked set and that all processes following P in the query trace list are also members of the minimal deadlocked set.

Theorem 5: If a process P in a minimal deadlocked set receives an engaging query from a process Q that is not in the minimal deadlocked set and subsequently replies to Q, then P will declare deadlock for itself and all members of its reachable set. Furthermore, P recognizes that its reachable set forms a minimal deadlocked set. When Q receives the reply from P, Q knows that Q itself is not in a minimal deadlocked set, and that the reachable set of P does comprise a minimal deadlocked set.

Proof: The proof for theorem 5 is similar to that for theorem 4.

Theorem 6: If a process P declares deadlock, it knows the identities of all members of every minimal deadlocked set in its reachable set. For each of these minimal deadlocked sets, there is a process Q belonging to the minimal deadlocked set such that Q knows the identity of the reachable set of Q, and that the reachable set of Q forms a minimal deadlocked set.

Proof: The proof for theorem 6 is similar to that for theorem 4.

CHAPTER 5.

Simulation Studies

5.1. Introduction

The CMH Algorithm and the five new algorithms proposed in this thesis were simulated to compare their performance and efficiencies. In these studies, the efficiency of an algorithm is measured in terms of the number and length of messages sent on the communication network, the additional storage required of each process, and the time required by the algorithms to detect deadlock once deadlock exists.

5.2. Design

The algorithms were simulated using six Pascal programs, with each program implementing one of the deadlock detection algorithms. The simulation is event–driven and uses an events list which holds events to occur in the future. As an event is forecast, it is added to the events list after all events already scheduled to occur earlier or at the same time. Events in the events list are processed one at a time. The simulation ends when the events list expires, or when a termination event is encountered.

5.3. Simulation Model of Communicating Processes

In the simulation model of communicating processes, an idle process begins to execute upon receiving a message from any process in its dependent set. An executing process may send a message to another process.

The time t between the instant at which a process P last became executing to the instant at which it will either send a message or become idle is exponentially distributed with mean $1/\lambda$. Let π be the probability that the process will send a message rather than becoming idle. After sending a message, an executing process will continue to execute for a period of time, after which it will become idle or send another message and continue executing. Let t_k be the length of time between the instant at which a process sends the $(k-1)$th message after it last became executing to the instant at which it will either become idle or send the kth message. We assume that the t_k's are statistically independent, identically distributed with average $1/\lambda$. When the probability π_k that the process will send the kth message is π independent of k, the length of time between the instant at which a process last became executing to the instant at which it will become idle is exponentially distributed with average $1/(\lambda\pi)$.

An executing process enqueues all messages it receives while executing. Upon becoming idle, it checks the queue for messages it may have received from processes in its dependent set. These messages are considered to be *acceptable* in that the process may accept one and begin executing again. Messages received by a process but not yet accepted for processing remain in the queue. If no acceptable messages have arrived, the process remains idle waiting for a message from any process in its dependent set. As an option of the simulation programs, however, messages may be discarded if they cannot be accepted immediately.

The model described here allows us to simulate direct communicating processes as well as a resource allocation system where a resource manager process is waiting to receive a message from a process releasing one of its resources held by that process. While the resource manager is waiting to regain control of its resources, other processes can request its resources by sending request messages to it. These messages are simply enqueued since the manager process does not have the requesting processes in its de; endent set.

5.4. Connection Topology

The processes form several clusters as shown in the schematic diagram in Figure 4. Such a model can be used to simulate a wide-area network, with each cluster in the model simulating a site or a local-area network within the wide-area network. The model can also be used to simulate a local-area network with different hosts. Communication between any two processes in the same cluster is subject only to an intra-cluster delay. Communication between two processes in different clusters is subject to an inter-cluster delay. For the cases simulated in this thesis, we measured the communication delays in terms of the number of hops required. This assumes that message and query traffic delays are constant and therefore are not affected by such factors as volume of network traffic and capacity of transmission channels. The assumption of constant network delays was made to allow all deadlock detection algorithms to be simulated in an identical environment. Such an assumption may not be valid, especially for the CMH Algorithm (Ref ShSi85). The effects these two factors may have on the algorithms simulated are discussed later in this chapter.

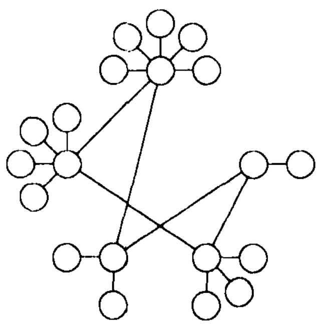


Figure 4
Clusters of Processes

5.5. Input Parameters

The following parameters are used to initialize the system.

(1) Number of clusters in the system.

(2) For each cluster, the number of processes in that cluster.

(3) For each cluster, the message delay time between two processes in that cluster.

(4) For each cluster, the message delay times to every other cluster.

(5) For each process, four parameters governing dependent set selection — When a process becomes idle, its new dependent set is formed and consists of some processes selected from within the same cluster and some processes selected from the other clusters in the network. The four parameters are the minimum and maximum number of processes in the dependent set which may come from the same cluster and the minimum and maximum number of processes in the dependent set which may come from all the other clusters. The new dependent set is chosen randomly by picking N_1 processes from the other clusters and N_2 processes from the same cluster. The numbers N_1 and N_2 are uniformly distributed between their respective minima and maxima (inclusive).

(6) Probability π that the next event of an executing process is to send another message and continue executing instead of becoming idle. This is a parameter which has associated with it two constraints. The first is a minimum number of processes which remain executing, and the second is a maximum number of processes which should be allowed to execute. If there are N processes in the system and the two constraints are 0 and N, then the constraints have no effect on the simulation. If,

however, the lower constraint is greater than 0, then a minimum number of processes will always be executing (that is, a process is not allowed to become idle if the minimum number of executing processes is not maintained).

(7) Average time $1/\lambda$ an executing process executes until its next event occurs.

(8) Number of messages used to initialize the system. The simulation is started with each process having just become idle and having generated a new dependent set. Messages are introduced into the system as though they had been sent at time 0, but not yet received. Processes will begin executing once they receive these messages.

5.6. Statistics Collected

Each *query*, *reply*, *cancel*, and *inform* contains the initiator identifier of the query computation and a sequence number. For Algorithms II, III, IV, and V, this sequence number is the sequence number associated with the query computation initiator. Each process has two categories of query computations associated with it: those that fail to detect deadlock, and one (or less) which succeeds in detecting deadlock. *Queries*, *replies*, *cancels*, and *informs* are tallied according to the query computation initiator and sequence number, with all old query computations for a given initiator lumped together. If deadlock is discovered, the contents of all counters are recorded. These counts show how much effort has been spent to that point.

In addition to the statistics mentioned above, the numbers of *queries*, *replies*, *cancels*, *informs*, and *messages* sent and accepted, both until the first deadlock is discovered and over the duration of the simulation, are recorded. The number of *messages* sent does not always equal the number of *messages* accepted, since a *message* is not be accepted by a process unless its intended receiver is idle and the sender is in the receiver's dependent set.

The total number of data fields required to be sent is also recorded. For the new algorithms which use a variable-length format, it is assumed that one extra field is required to indicate the length of the *query*, *reply*, *cancel*, or *inform*. Header information was not added to the number of data fields required.

Associated with each process is a variable showing when that process last changed state from executing to idle. Since our algorithms identify all processes in the deadlocked set, it is possible to determine the time at which the deadlock first developed. This information is then used in determining how long the algorithms require to detect deadlock.

Two additional statistics collected are the average number of idle processes and the average size of dependent sets. The average size of dependent sets ($\hat{D}$) is a time-weighted average which takes into account the durations of dependent sets. Thus, if during the course of the simulation of duration T, there are m dependent sets generated for all processes, and dependent set i has size S_i and duration t_i. then

$$\hat{D} = \frac{\sum_{i=1}^{m}(t_i * S_i)}{\sum_{i=1}^{m} t_i} \tag{1}$$

The average number of idle processes ($\hat{N}$) in the system is also a time weighted average which takes into account the amount of time a process is idle.

$$\hat{N} = \frac{\sum_{i=1}^{m} t_i}{T} \tag{2}$$

The average number of dependency relationships ($\hat{C}$) is the product of the average size of a dependent set ($\hat{D}$) and the average number of idle processes in the system ($\hat{N}$).

$$\hat{C} = \hat{D} * \hat{N} \tag{3}$$

5.7. Results and Analysis

The first case study is carried out to determine in which order an idle process should query the processes in its dependent set. Case studies 2 and 3 are done in order to fix the values of T_1 and T_2 for the remaining case studies. In case studies 4 through 7, the performance of the algorithms are evaluated.

5.7.1. Case Study 1

The first case study of communicating processes demonstrates that, for Algorithms III, IV, and V, processes in a dependent set should be queried in the order of highest priority first, rather than lowest priority first. Table 1 shows the input parameters used in this case study. Each test configuration was run 75 times. The first test configuration was designed so that all 75 runs resulted in detection of deadlock, and an average of 64 messages were sent during each run. The second test configuration was designed so that at least 2 processes were executing at any one time, and deadlock was never detected in all 75 runs. Each run executed for 5000 time units, and an average of 1481 messages were sent during each run. The means and the 90% confidence intervals for the two test configurations are shown in Tables 2 and 3.

Tables 2 and 3 show that it is important that processes be queried in the order of highest priority first, especially when deadlock is likely to occur. For the remainder of the simulation studies, processes in a dependent set are queried in order of highest priority first for Algorithms III, IV, and V.

Table 1
Input Parameters for Case Study 1 of Communicating Processes

Parameter	With Deadlock	Without Deadlock
Number of processes in cluster 1	3	3
Number of processes in cluster 2	4	4
Number of processes in cluster 3	3	3
Intra-, Inter-cluster delay times	2, 5	2, 5
Mean time a process executes until next event	10	10
Probability a process sends a message for its next event (constraints)	0.50 (0,10)	0.50 (2,10)
Number of processes in dependent set from same cluster	1–2	1–2
Number of processes in dependent set from other clusters	1–5	4–6
Times T_1 and T_2 a process waits before processing query computations	50, 45	50, 45

Table 2
Priority Comparison With Deadlock

Algorithm	Volume of Query Traffic with Deadlock Detected			
	highest priority first		lowest priority first	
	mean	90% confidence	mean	90% confidence
Algorithm III	189.013	(178.436 , 199.590)	359.693	(345.665 , 373.772)
Algorithm IV	111.680	(101.428 , 121.932)	157.733	(147.342 , 168.125)
Algorithm V	111.667	(101.414 , 121.919)	157.227	(146.802 , 167.651)

AD-A172 839 DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING PROCESSES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH R A ADAMS 1986 AFIT/CI/NR-86-174D 2/3

UNCLASSIFIED F/G 9/2 NL

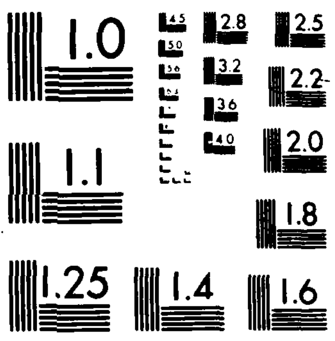

1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Table 3
Priority Comparison Without Deadlock

Algorithm	Volume of Query Traffic without Detecting Deadlock			
	highest priority first		lowest priority first	
	mean	90% confidence	mean	90% confidence
Algorithm III	287.880	(270.923 , 304.837)	312.987	(295.639 , 330.334)
Algorithm IV	282.293	(264.489 , 300.098)	301.760	(283.525 , 319.995)
Algorithm V	282.507	(264.679 , 300.334)	301.427	(283.196 , 319.658)

5.7.2. Case Study 2

T_1 is the time a process waits after becoming idle before initiating a query computation. T_2 is the minimum time after a process becomes idle before it sends to a process in its dependent set a query in a query computation initiated by another process. The result of this case study shows the efficiency of each algorithm is very sensitive to the parameter T_1 for any given set of input parameters. Table 4 lists the different parameter values used in the first three test configurations, each of which contains a single large cluster of processes. Table 5 lists the parameter values for three more test configurations containing five small clusters of processes. For this case study, T_2 was taken to be $0.9 * T_1$. Different values of T_2 are used in case study 3. The results are shown in Figures 5 through 10 for the first three test configurations, and in Figures 11 through 16 for the second three test configurations. Each line on the graphs shows the total amount of query traffic (*queries*, *replies*, and *cancels*). Because the simulation may be halted with the events list still containing future events, an error is introduced into the measured result. The size of this error is dependent on T_1 and T_2. For example, if $T_1 =$ 500, and the duration of the simulation is 5000, then any process becoming idle after 4500 does not affect the amount of query taffic passed before the simulation halts, since the query computation for that process will not start until after the simulation halts. In this case, the duration of the simulation (5000) is long enough to ensure that the error introduced in this manner is less than approximately 10%. For the value $T_1 = 800$, the simulation was run for 10,000 time units, with the volume of query traffic being divided by 2. Thus, the shape of the curve approximately reflects the amount of query traffic which will never detect deadlock. For Algorithms CMH, I, and II, each test configuration was simulated enough times so that we can say that the expected mean lies within 10% of the sample mean with an 80% level of confidence. For Algorithms III, IV, and V, the same can be said with a 90% level of confidence.

Table 4
Input Parameters for Case Study 2 Using One Large Cluster

Parameter	Test Configuration 1 Small Size Dependent Sets	Test Configuration 2 Medium Size Dependent Sets	Test Configuration 3 Large Size Dependent Sets
Processes in cluster 1	20	20	20
Intra-cluster delay times (in hops)	2	2	2
Mean time a process executes until next event	20	20	20
Probability a process sends a message for its next event (constraints)	0.50 (3,20)	0.50 (3,20)	0.50 (3,20)
Processes in dependent set	1-8	9-14	15-19
Time T_1 a process waits before initiating a query computation	100, 200 350, 500 800	100, 200 350, 500 800	100, 200 350, 500 800

Table 5
Input Parameters for Case Study 2 Using Five Small Clusters

Parameter	Test Configuration 4 Small Size Dependent Sets	Test Configuration 5 Medium Size Dependent Sets	Test Configuration 6 Large Size Dependent Sets
Processes in cluster 1	4	4	4
Processes in cluster 2	3	3	3
Processes in cluster 3	4	4	4
Processes in cluster 4	5	5	5
Processes in cluster 5	4	4	4
Intra-, Inter-cluster delay times (in hops)	2, 5	2, 5	2, 5
Mean time a process executes until next event	20	20	20
Probability a process sends a message for its next event (constraints)	0.50 (3,20)	0.50 (3,20)	0.50 (3,20)
Processes in dependent set from same cluster	1-2	1-3	2-4
Processes in dependent set from other clusters	1-5	5-10	10-15
Time T_1 a process waits before initiating a query computation	100, 200 350, 500 800	100, 200 350, 500 800	100, 200 350, 500 800

As T_1 becomes smaller, an idle process initiates a query computation sooner, and therefore has a greater chance of initiating a query computation which encounters an executing process. There is also a greater chance that the initiator will become executing after initiating the query computation. Plots in Figures 5 through 16 show what happens as the average size of the dependent sets increases. For Algorithm CMH, the volume of query traffic increases due to the multiplying effect caused by the breadth-first approach. For the depth-first algorithms, however, just the opposite occurs. The results of our simulation show that as the average size of the dependent sets increases, the average number of idle processes encountered by queries in a query computation actually decreases. Hence, a query computation using the depth-first approach tends to encounter an executing process sooner, and the average volume of query traffic is less.

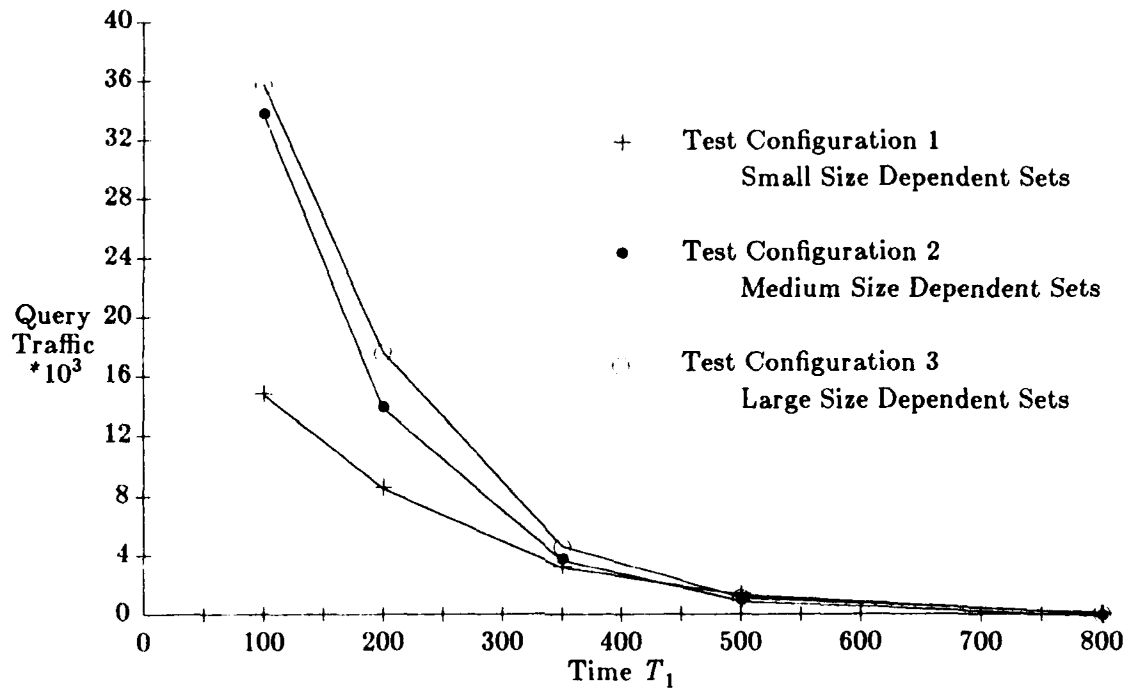


Figure 5
Sensitivity of Algorithm CMH to T_1
For Configurations Containing One Large Cluster

With the exception of case study 3, the rest of our results were obtained with the value of T_1 chosen to be 500. This value of T_1 is chosen because for values less than 500, the CMH Algorithm requires significantly more query traffic. For our algorithms, increasing T_1 from 500 to 800 resulted in no significant decrease in query traffic. For the second set of plots in which the processes are in five small clusters, we observed similar results showing the CMH Algorithm requiring significantly more query traffic for values of T_1 less than 500, and no significant improvement for our algorithms when increasing T_1 from 500 to 800. We observe that the performance of the algorithms does not depend significantly on whether the processes form one or more than one cluster when the inter-cluster delay times are not too much greater than the intra-cluster delay times.

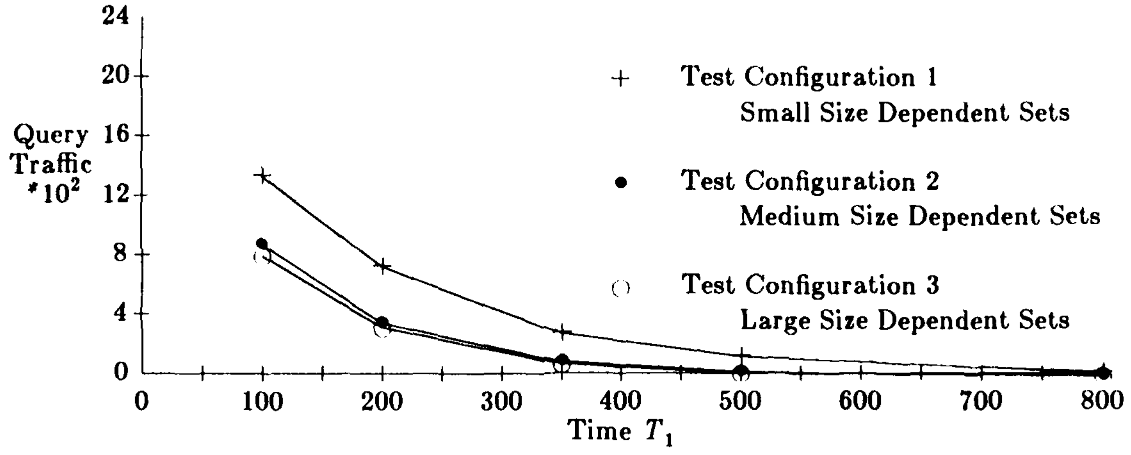


Figure 6
Sensitivity of Algorithm I to T_1
for Configurations Containing One Large Cluster

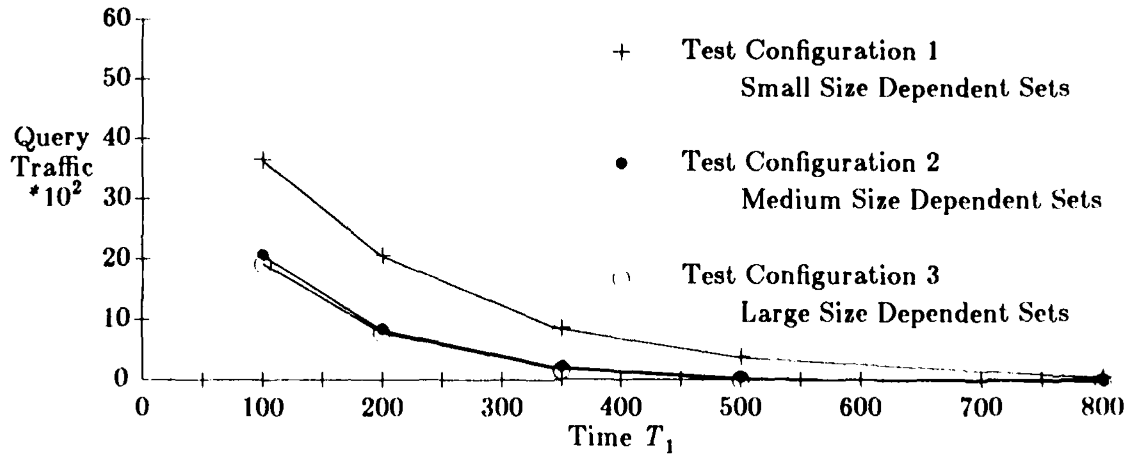


Figure 7
Sensitivity of Algorithm II to T_1
for Configurations Containing One Large Cluster

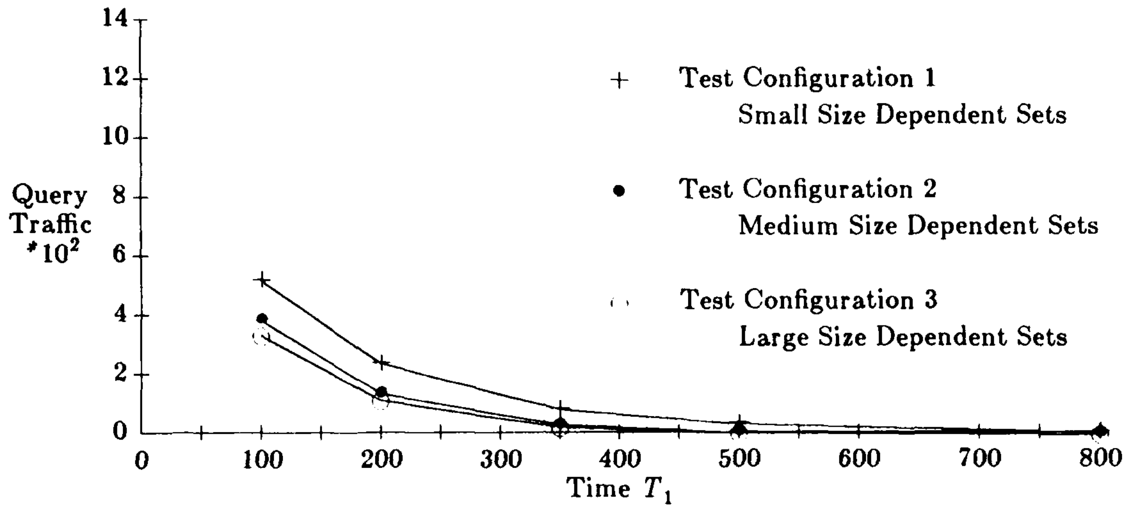


Figure 8
Sensitivity of Algorithm III to T_1
for Configurations Containing One Large Cluster

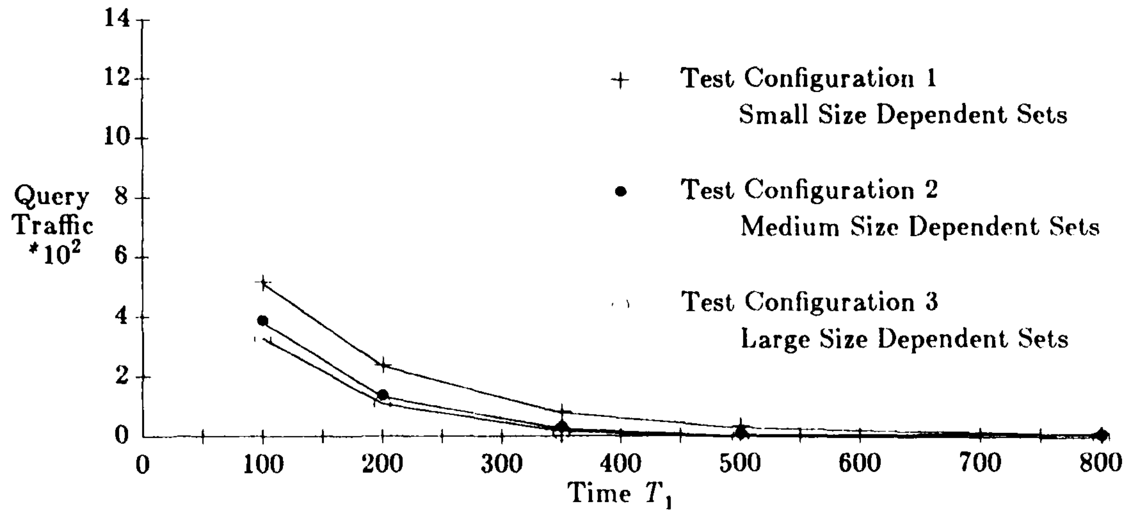


Figure 9
Sensitivity of Algorithm IV to T_1
for Configurations Containing One Large Cluster

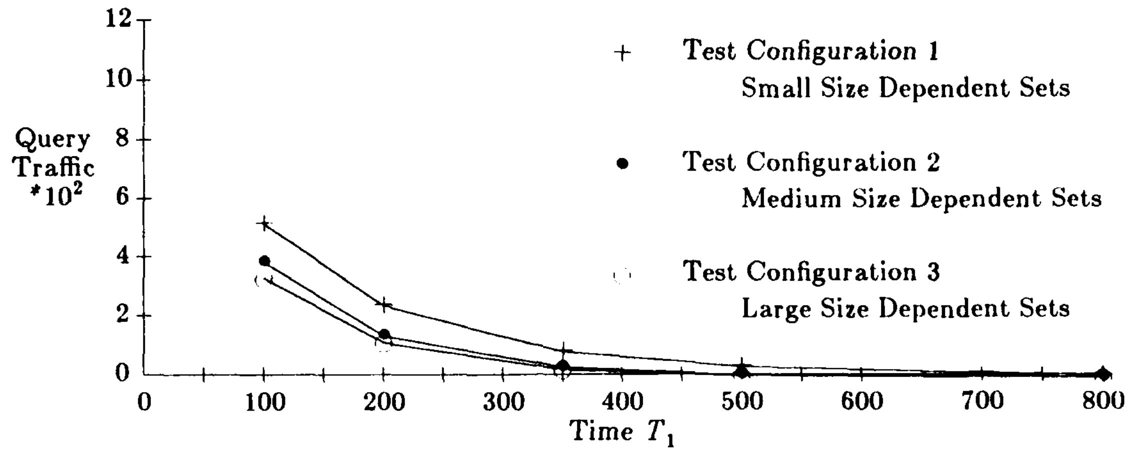


Figure 10
Sensitivity of Algorithm V to T_1
for Configurations Containing One Large Cluster

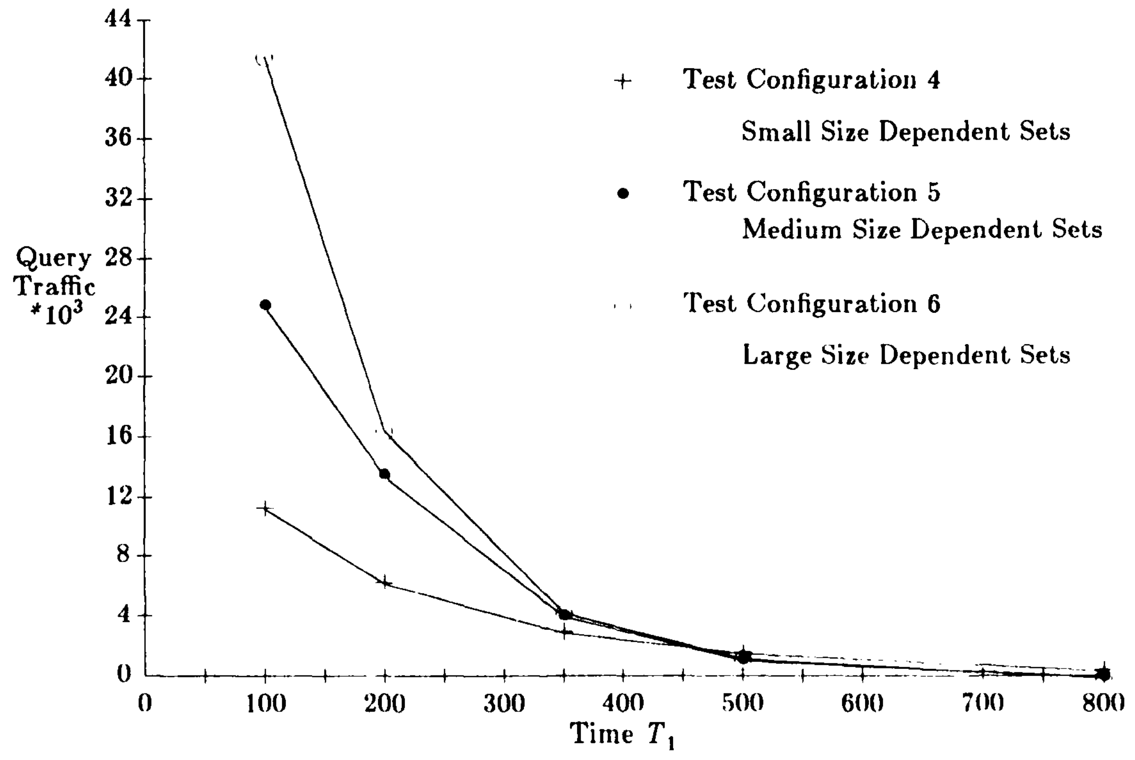


Figure 11
Sensitivity of Algorithm CMH to T_1
for Configurations Containing Five Small Clusters

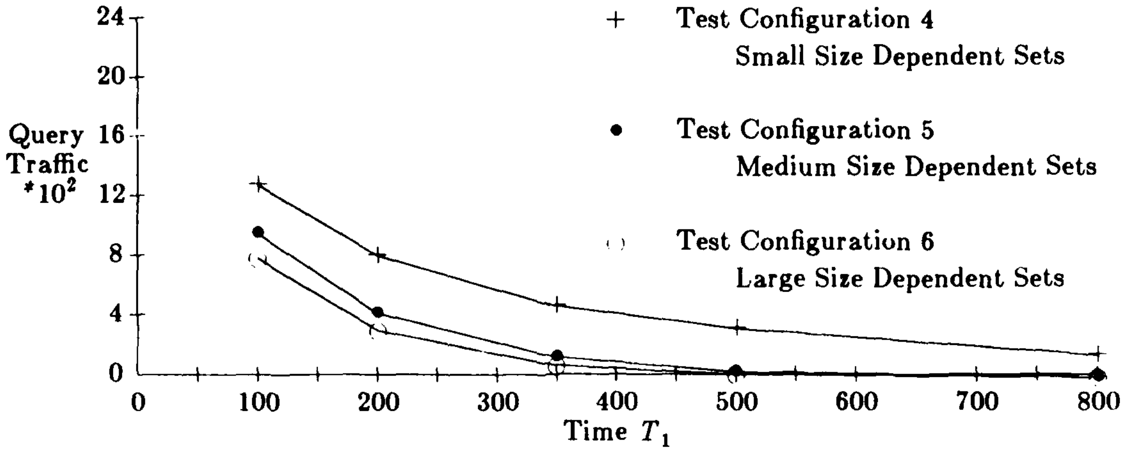


Figure 12
Sensitivity of Algorithm I to T_1
for Configurations Containing Five Small Clusters

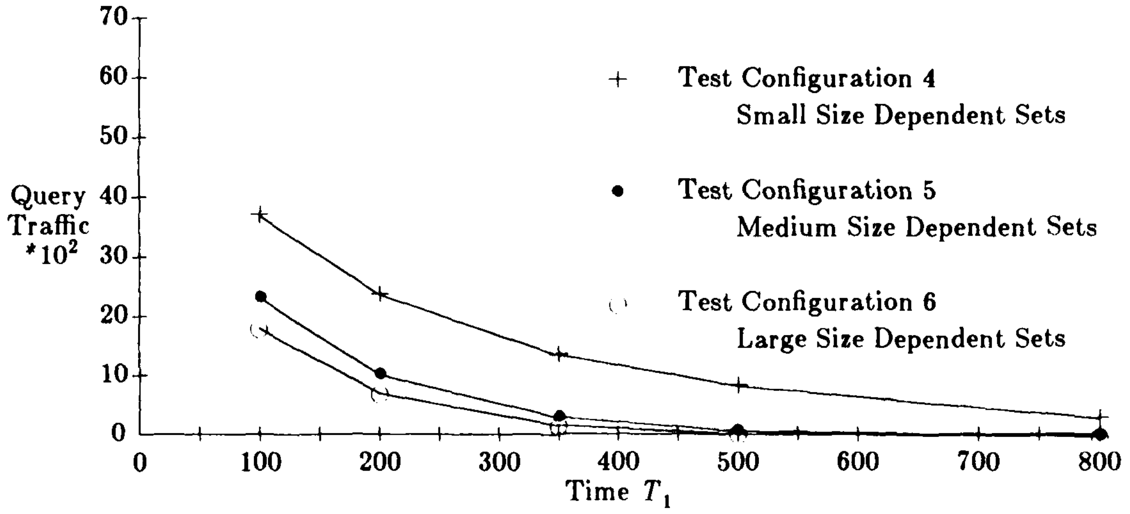


Figure 13
Sensitivity of Algorithm II to T_1
for Configurations Containing Five Small Clusters

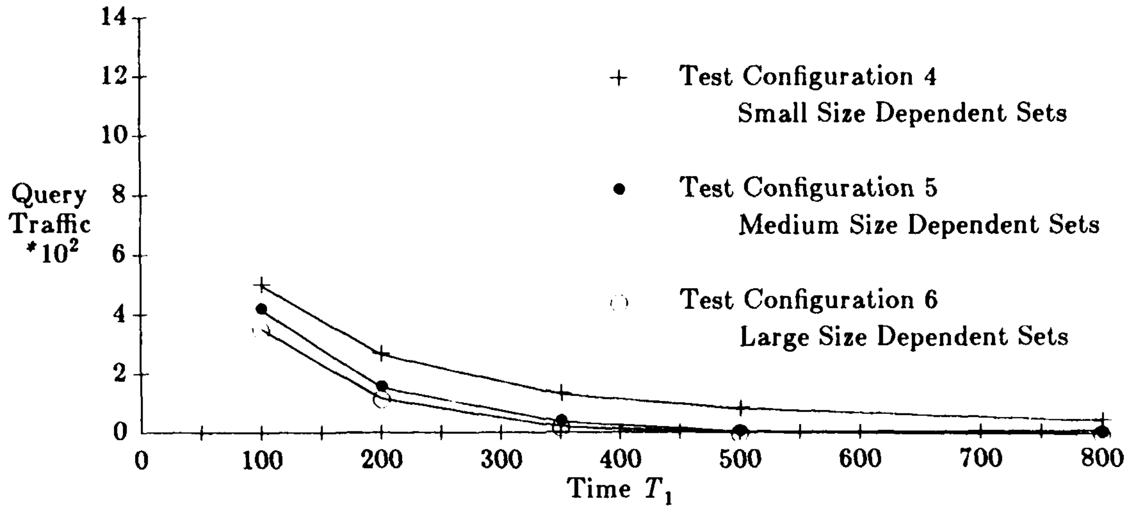


Figure 14
Sensitivity of Algorithm III to T_1
for Configurations Containing Five Small Clusters

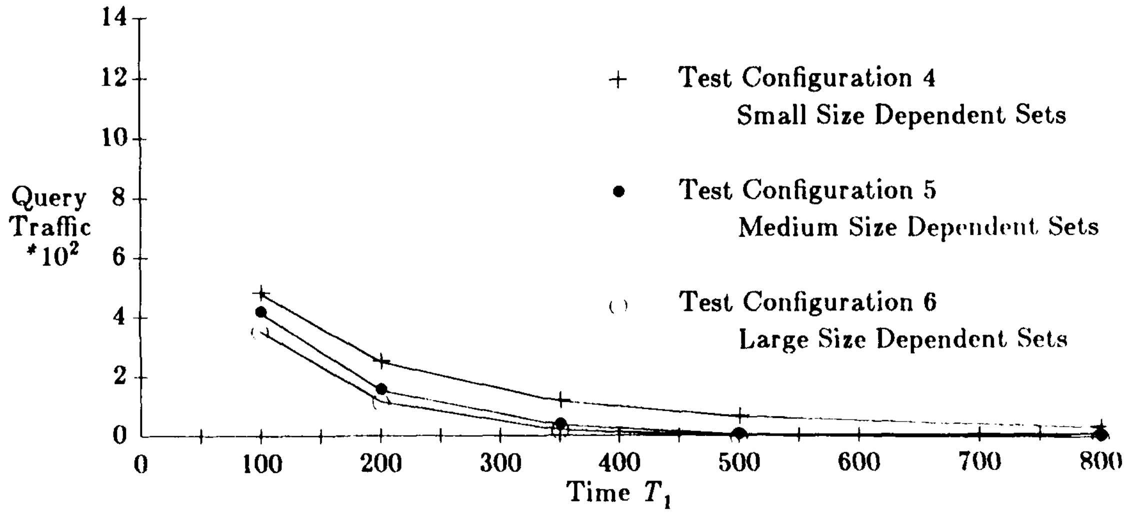


Figure 15
Sensitivity of Algorithm IV to T_1
for Configurations Containing Five Small Clusters

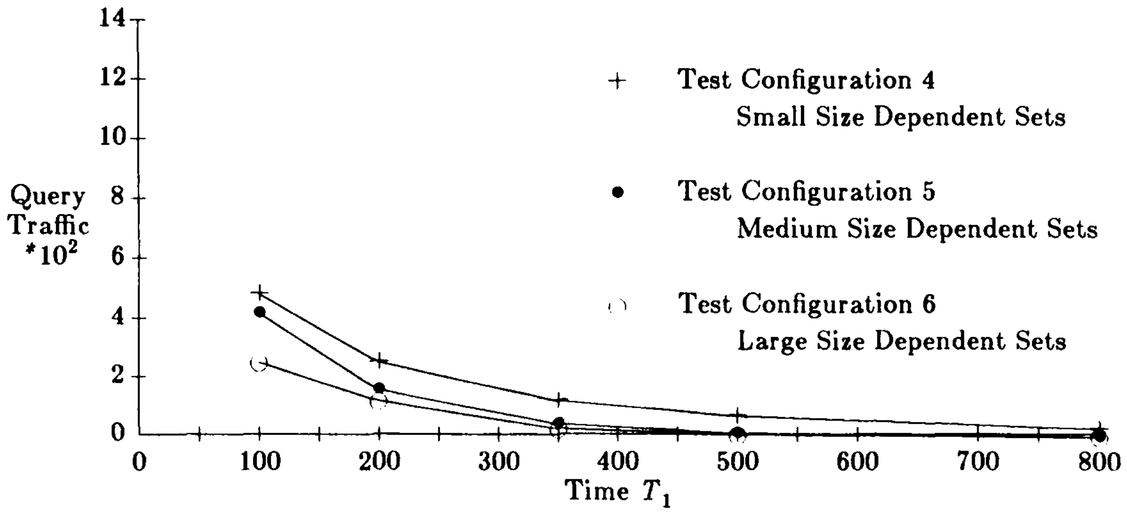


Figure 16
Sensitivity of Algorithm V to T_1
for Configurations Containing Five Small Clusters

5.7.3. Case Study 3

This case study shows that the performance of Algorithms III, IV, and V is sensitive to the parameter T_2 for any given set of other input parameters. Test configurations 1, 2, 3, and 5 from case study 2 were selected, and T_2 was assigned the values 0.9 T_1, 0.5 T_1, and 0.1 T_1. Table 6 shows the different parameter values used in this simulation. Figures 17 through 22 show the results for the three test configurations containing a single large cluster, and Figures 23 through 24 show the results for the test configuration with five small clusters. Since the data for Algorithms IV and V were the same, their plots were combined. Each line on the graphs shows the total amount of query traffic (*queries*, *replies*, and *cancels*) averaged over 50 trials. As in case study 2, we have an 80% level of confidence that the expected mean lies within 10% of the sample mean. For Algorithms III, IV, and V, the same can be said with a 90% level of confidence.

Table 6
Input Parameters for Case Study 3

Parameter	Test Config. 1 One Large Cluster Small Size Dependent Sets	Test Config. 2 One Large Cluster Medium Size Dependent Sets	Test Config. 3 One Large Cluster Large Size Dependent Sets	Test Config. 5 Five Small Clusters Medium Size Dependent Sets
Processes in cluster 1	20	20	20	4
Processes in cluster 2	0	0	0	3
Processes in cluster 3	0	0	0	4
Processes in cluster 4	0	0	0	5
Processes in cluster 5	0	0	0	4
Intra-, Inter-cluster delay times (in hops)	2	2	2	2, 5
Mean time a process executes until its next event	20	20	20	20
Probability a process sends a message for its next event (constraints)	0.50 (3,20)	0.50 (3,20)	0.50 (3,20)	0.50 (3,20)
Number of processes in dependent set from same cluster	3-8	9-14	15-19	1-3
Number of processes in dependent set from other clusters	0	0	0	5-10
Time T_1 a process waits before initiating a query computation	100, 200 350, 500 800	100, 200 350, 500 800	100, 200 350, 500 800	100, 200 350, 500 800
Time T_2 a process waits before processing other query computations	$0.1\ T_1$ $0.5\ T_1$ $0.9\ T_1$	$0.1\ T_1$ $0.5\ T_1$ $0.9\ T_1$	$0.1\ T_1$ $0.5\ T_1$ $0.9\ T_1$	$0.1\ T_1$ $0.5\ T_1$ $0.9\ T_1$

T_2 is the minimum time a process waits after becoming idle before it sends to a process in its dependent set a query in a query computation initiated by another process. The plots in Figures 17 through 24 show that as T_2 increases from $0.1\ T_1$ to $0.9\ T_1$, the volume of query traffic is approximately cut in half. For values of T_1 greater than 200, very little is gained by increasing T_2 from $0.5\ T_1$ to $0.9\ T_1$. Since in case study 2 we chose T_1 to be 500, any choice of T_2 greater than 0.5 T_1 does not significantly affect the volume of query traffic. As T_2 increases, the time to detect deadlock tends to increase, since the last process to become idle waits longer before processing any query computations. We choose $T_2 = 0.9\ T_1$ for the remaining case studies primarily to minimize the volume of query traffic.

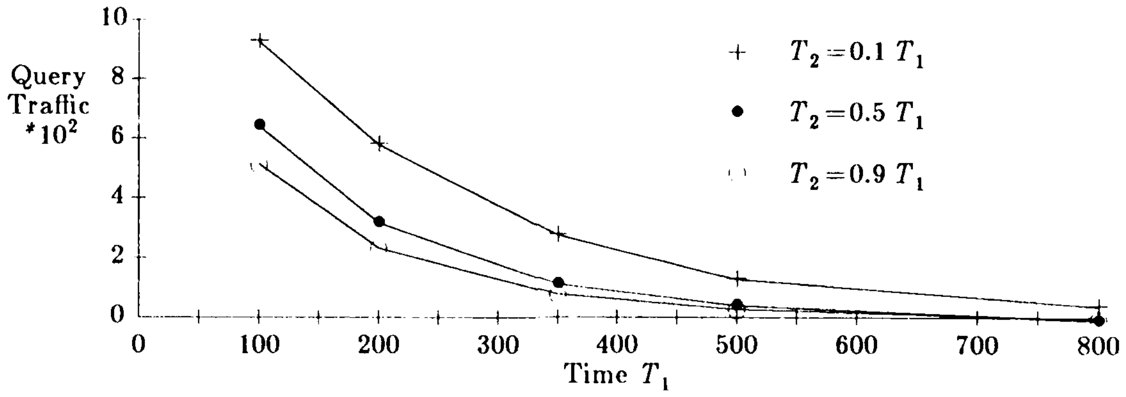


Figure 17
Sensitivity of Algorithm III to T_2, Test Configuration 1
With One Large Cluster and Small Size Dependent Sets

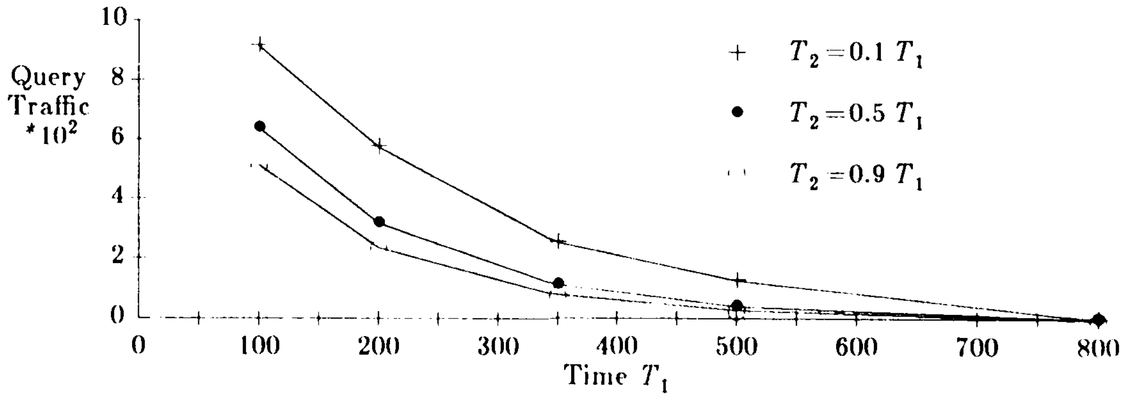


Figure 18
Sensitivity of Algorithms IV and V to T_2, Test Configuration 1
With One Large Cluster and Small Size Dependent Sets

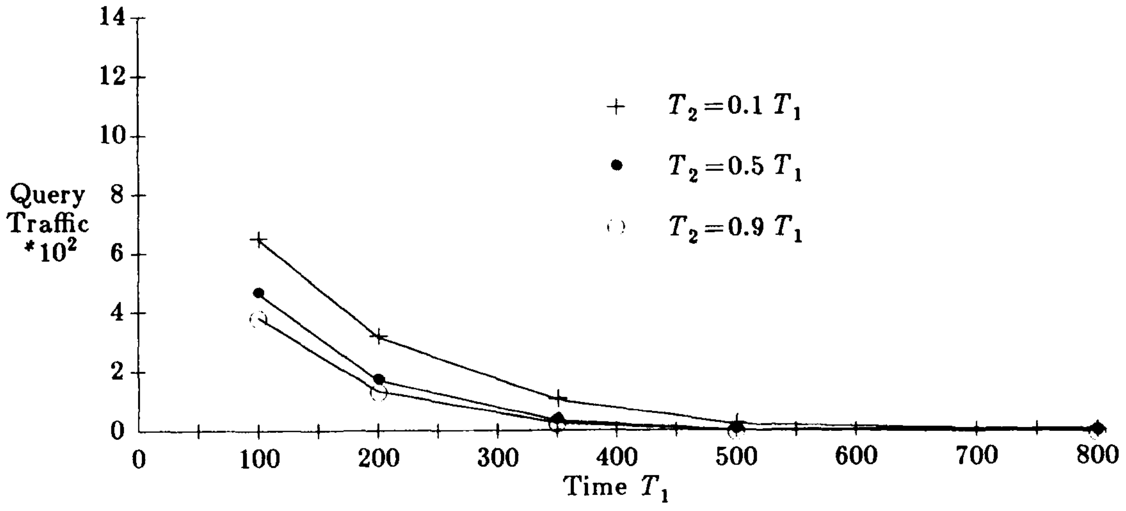


Figure 19
Sensitivity of Algorithm III to T_2, Test Configuration 2
With One Large Cluster and Medium Size Dependent Sets

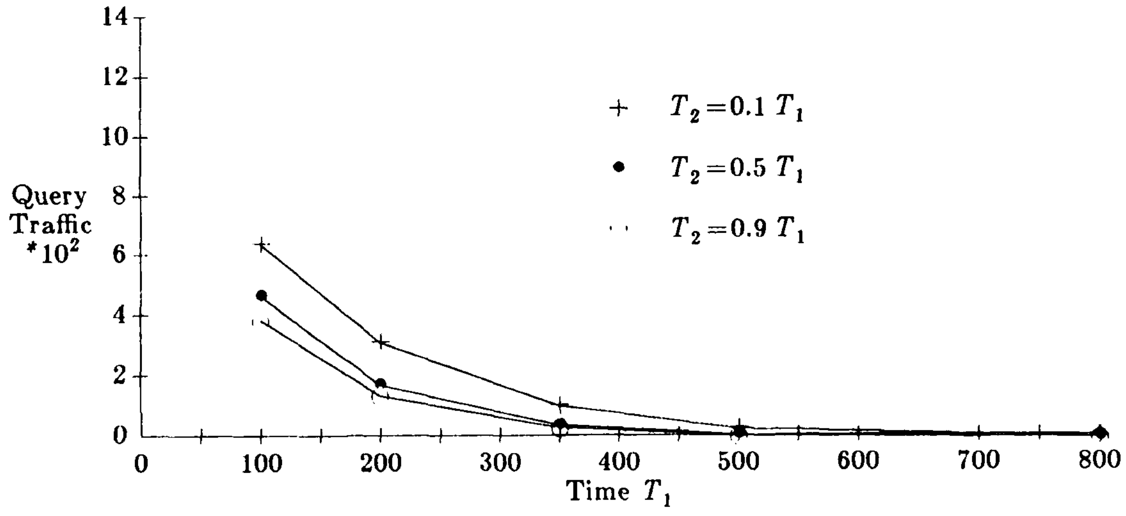


Figure 20
Sensitivity of Algorithms IV and V to T_2, Test Configuration 2
With One Large Cluster and Medium Size Dependent Sets

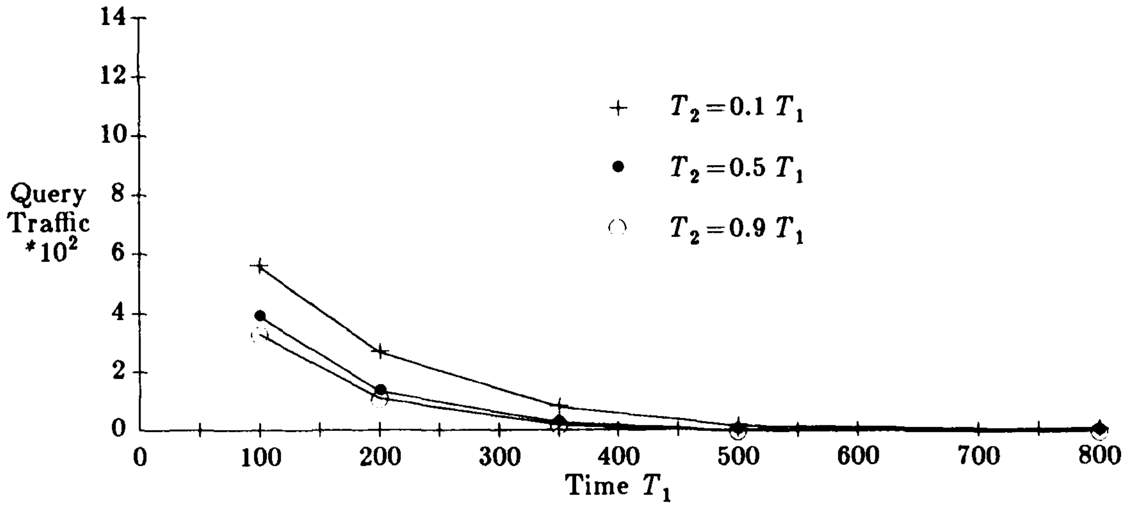


Figure 21
Sensitivity of Algorithm III to T_2, Test Configuration 3
With One Large Cluster and Large Size Dependent Sets

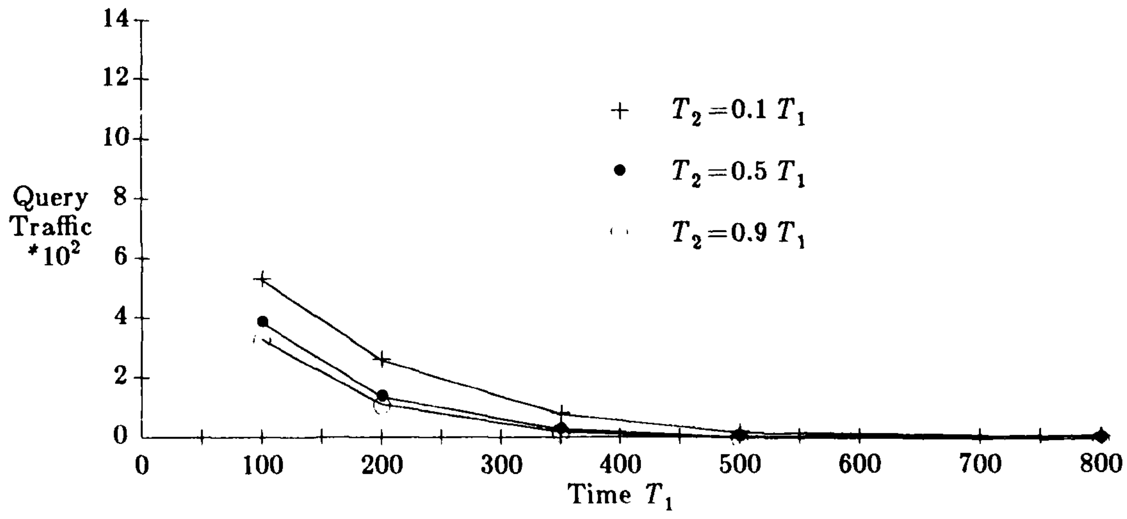


Figure 22
Sensitivity of Algorithms IV and V to T_2, Test Configuration 3
With One Large Cluster and Large Size Dependent Sets

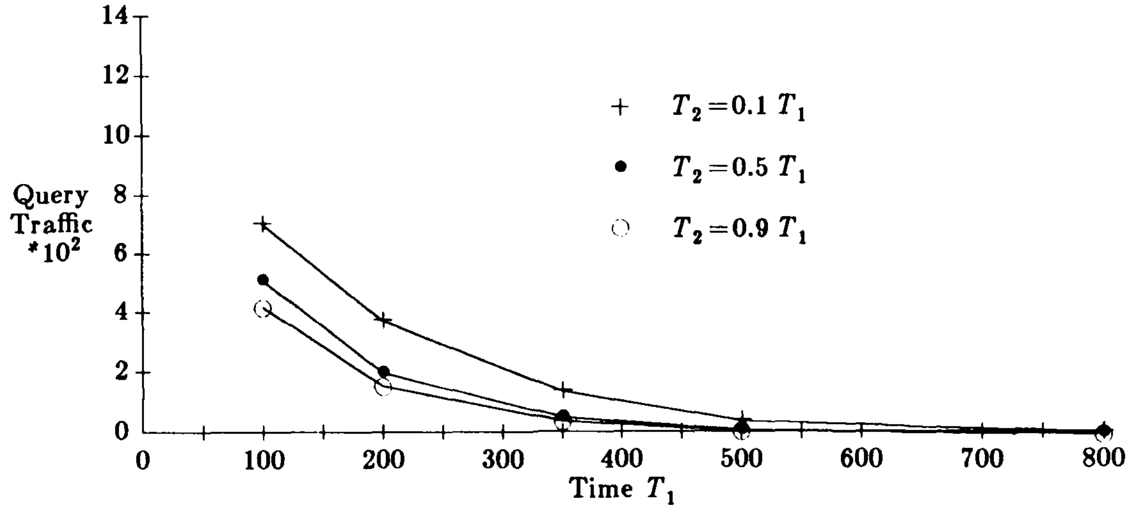


Figure 23
Sensitivity of Algorithm III to T_2, Test Configuration 5
With Five Small Clusters and Medium Size Dependent Sets

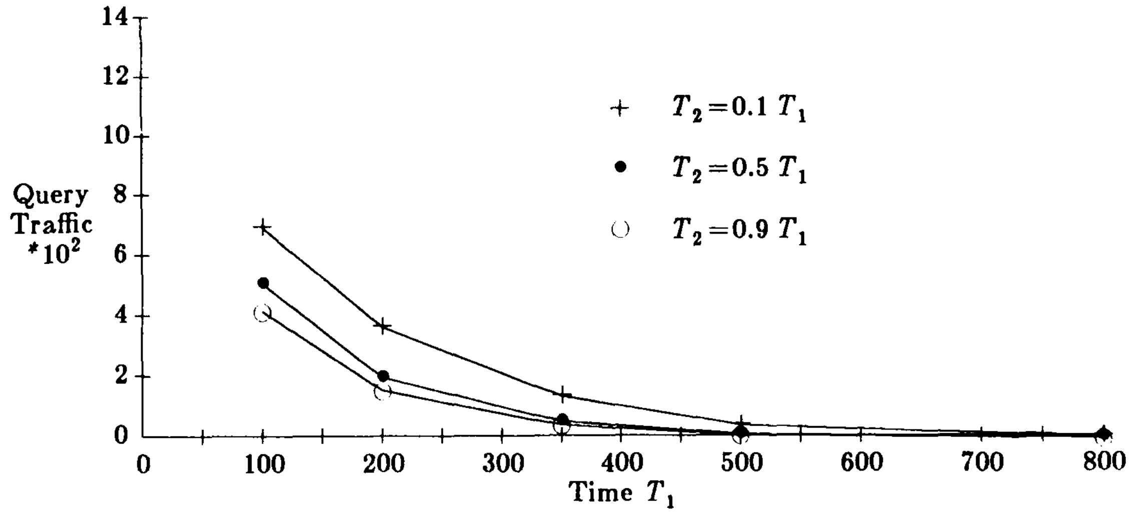


Figure 24
Sensitivity of Algorithms IV and V to T_2, Test Configuration 5
With Five Small Clusters and Medium Size Dependent Sets

5.7.4. Case Study 4

The fourth case study evaluates the time required to detect deadlock once deadlock exists and the load placed on the network during deadlock detection. The input parameters for the two test configurations used in this example are shown in Table 7. The values of T_1 and T_2 are chosen to be 500 and 450 and are based on the results from case studies 2 and 3. Each test configuration was simulated to obtain 25 times when deadlock existed. This was sufficient to establish that, with an 80% level of confidence, the expected mean differs from the sample mean by no more than 15% of the sample mean. The performance characteristics for the algorithms when deadlock exists are shown in Figures 25 and 26. For each run R_i, the time when the last process in the deadlocked became permanently idle is Ψ_i. That is, deadlock first existed at time Ψ_i. The time scales for each run were normalized with respect to the largest Ψ_i, which was 439. The statistics were collected in buckets, with each bucket collecting data from all 20 processes for 50 units of time. The results were averaged over the 25 trials.

For Test Configuration 1, the first process became permanently idle at time 300, and the number of permanently idle processes increased until at time 439, all processes had become permanently idle. Since $T_1 = 500$, any query computation begun after time 939 will result in detection of deadlock. For the CMH Algorithm curve, the portion between time 800 and time 950 represents mostly query computation traffic for which deadlock is not detected.

The CMH Algorithm exhibits a large surge of query computation traffic since it employs a breadth-first method. Assuming the system can handle this surge with no increase in message delivery time, the CMH Algorithm will complete faster than the other

5.7.4. Case Study 4

The fourth case study evaluates the time required to detect deadlock once deadlock exists and the load placed on the network during deadlock detection. The input parameters for the two test configurations used in this example are shown in Table 7. The values of T_1 and T_2 are chosen to be 500 and 450 and are based on the results from case studies 2 and 3. Each test configuration was simulated to obtain 25 times when deadlock existed. This was sufficient to establish that, with an 80% level of confidence, the expected mean differs from the sample mean by no more than 15% of the sample mean. The performance characteristics for the algorithms when deadlock exists are shown in Figures 25 and 26. For each run R_i, the time when the last process in the deadlocked became permanently idle is Ψ_i. That is, deadlock first existed at time Ψ_i. The time scales for each run were normalized with respect to the largest Ψ_i, which was 439. The statistics were collected in buckets, with each bucket collecting data from all 20 processes for 50 units of time. The results were averaged over the 25 trials.

For Test Configuration 1, the first process became permanently idle at time 300, and the number of permanently idle processes increased until at time 439, all processes had become permanently idle. Since $T_1 = 500$, any query computation begun after time 939 will result in detection of deadlock. For the CMH Algorithm curve, the portion between time 800 and time 950 represents mostly query computation traffic for which deadlock is not detected.

The CMH Algorithm exhibits a large surge of query computation traffic since it employs a breadth-first method. Assuming the system can handle this surge with no increase in message delivery time, the CMH Algorithm will complete faster than the other

Table 7
Input Parameters for Case Study 4

Parameter	Test Configuration 1 Small to Medium Dependent Sets	Test Configuration 2 Medium to Large Dependent Sets
Number of processes in cluster 1	4	4
Number of processes in cluster 2	3	3
Number of processes in cluster 3	4	4
Number of processes in cluster 4	5	5
Number of processes in cluster 5	4	4
Intra-, Inter-cluster delay times	2, 5	2, 5
Mean time a process executes until next event	20	20
Probability a process sends a message for its next event (constraints)	50 (0,20)	50 (0,20)
Number of processes in dependent set from same cluster	1-2	2-3
Number of processes in dependent set from other clusters	1-7	8-15
Times T_1 and T_2 a process waits before processing query computations	500,450	500,450

algorithms. However, the assumption of constant message and query traffic delays may not be valid, especially for the CMH Algorithm [Ref ShSi85]. Most communication networks cannot handle the initial surge without increasing message delivery time; effectively there is a ceiling on the network throughput which in turn limits the volume of query traffic on the network at any one time. For example, if the maximum throughput of the network for Test Configuration 1 is placed at 300 messages per time unit, then the CMH Algorithm requires approximately 1700 time units to detect deadlock, and Algorithms I and II are unaffected in Figure 25.

Algorithm II requires a greater volume of query computation traffic than Algorithm I because the query computations initiated before time 939 do not terminate when they encounter an executing process. Hence, there are always 20 query computations which

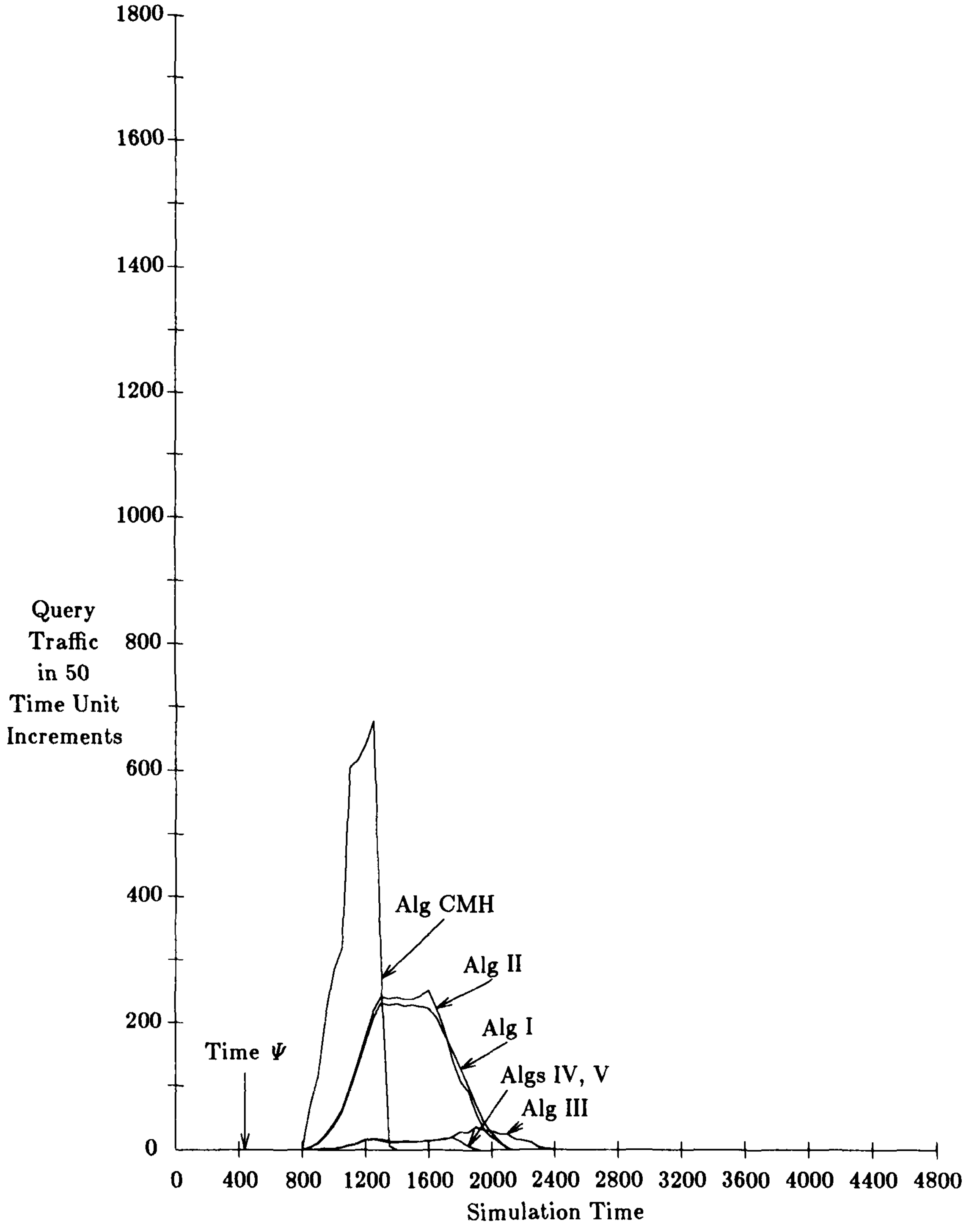


Figure 25

Time to Detection of Deadlock and System Load, Test Configuration 1 With Five Small Clusters and Small to Medium Size Dependent Sets

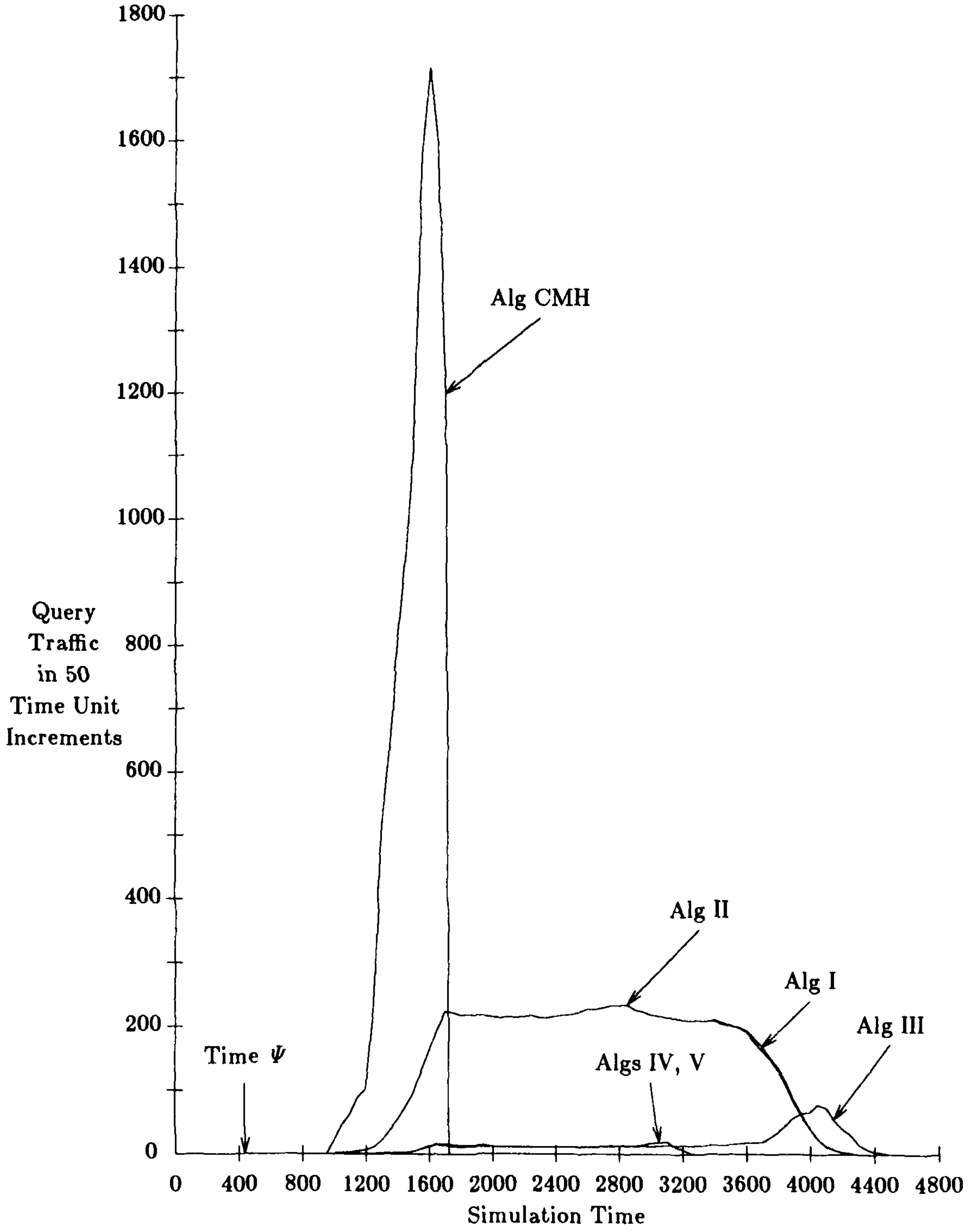


Figure 26

Time to Detection of Deadlock and System Load, Test Configuration 2 With Five Small Clusters and Medium to Large Size Dependent Sets detect deadlock. The heights of the curves for Algorithms I and II are dependent on the number of query computations and the message delay times, since for each query computation, there is only one *query* or *reply* in transit at any given time.

For Algorithms III, IV, and V in Test Configuration 1, deadlock is first detected around time 1750. The curves for Algorithms III, IV, and V show that the priority system is very effective in reducing query traffic for all but the highest priority query computation. The curves for Algorithms IV and V also show that using information from other query computations is very effective in reducing the volume of query traffic, especially when the highest priority query computation has completed. This result is explained further in case study 5. The slight surge in the volume of query traffic for Algorithm III after time 1750 shows the effect of other query computations processing once they are not blocked by higher priority ones.

The curves for Test Configuration 2 are similar in shape to those for Test Configuration 1. The average size of dependent sets for Test Configuration 2 is almost three times as large as for Test Configuration 1. Figures 25 and 26 show that the height of the CMH Algorithm curve is about 2.5 times as high, although not much wider. For the CMH Algorithm, the width of the curve (that is, the time to detect deadlock) is primarily determined by the height of the search tree and the message delay times. The deadlocks detected in this case study produced relatively short tree heights for the CMH Algorithm.

As in Test Configuration 1, the heights of the curves for Algorithms I and II are determined by the number of query computations in the system and the message delay times. Hence, the heights of these curves are approximately the same for both test configurations. The heights of the curves for Algorithms III, IV, and V are also approximately the same for both test configurations. The areas under the curves are greater, though, showing that as the number of dependency relationships increases, the time required for the depth-first approach to detect deadlock also increases.

5.6.5 Case Study 5

The fifth case study evaluates the volume of query traffic used by each process during deadlock detection. The input parameters for the two tests used are the same as for case study 4. Figures 27 and 28 show that the CMH Algorithm requires a larger volume of query traffic than Algorithms I and II. This happens because, in the CMH Algorithm, a process queries each member of its dependent set, whereas for Algorithms I and II, a process does not need to query its engager. This modification could be implemented very easily in the CMH Algorithm. The results for Algorithm III show most of the work being done by the highest priority query computation. For the fourth and successive query computations, the level of work required is approximately that required by the initiator to query each member of its dependent set and hence is related to the size of the dependent sets. For Algorithms IV and V, the highest priority query computation requires less query traffic than for Algorithm III because for Algorithms IV and V, it uses information from other query computations. Lower priority query computations in Algorithms IV and V require very little query traffic because they use information from previous query computations, and in particular, from the highest priority query computation. The volume of query traffic required by the lower priority query computations does not appear to depend on the size of the dependent sets or the number of processes in the system. On the average, the query computations initiated by processes 3 through 20 only require one *query* and one *reply* to determine they are deadlocked.

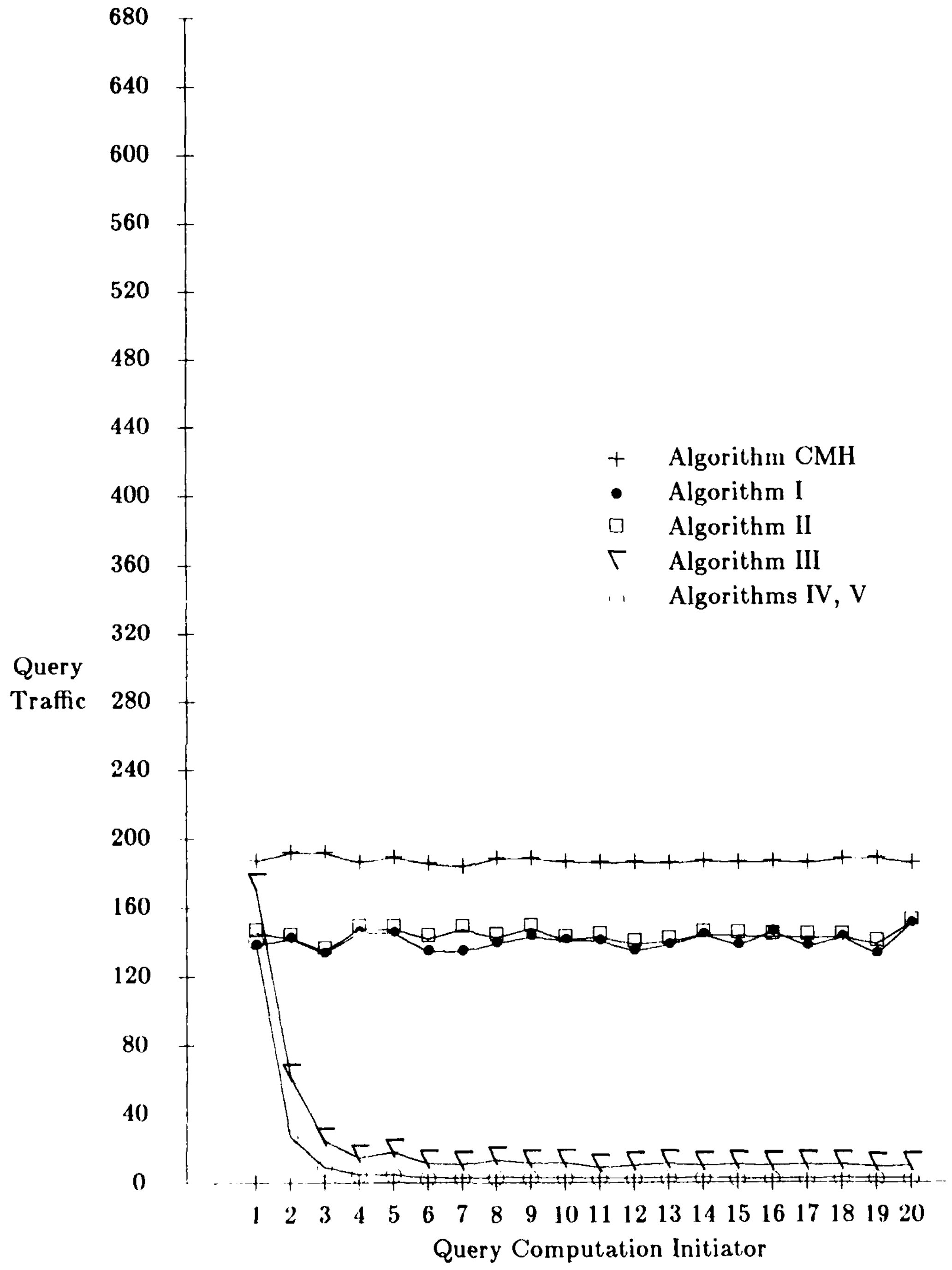


Figure 27

Volume of Query Traffic for Test Configuration 1

With Five Small Clusters and Small to Medium Size Dependent Sets

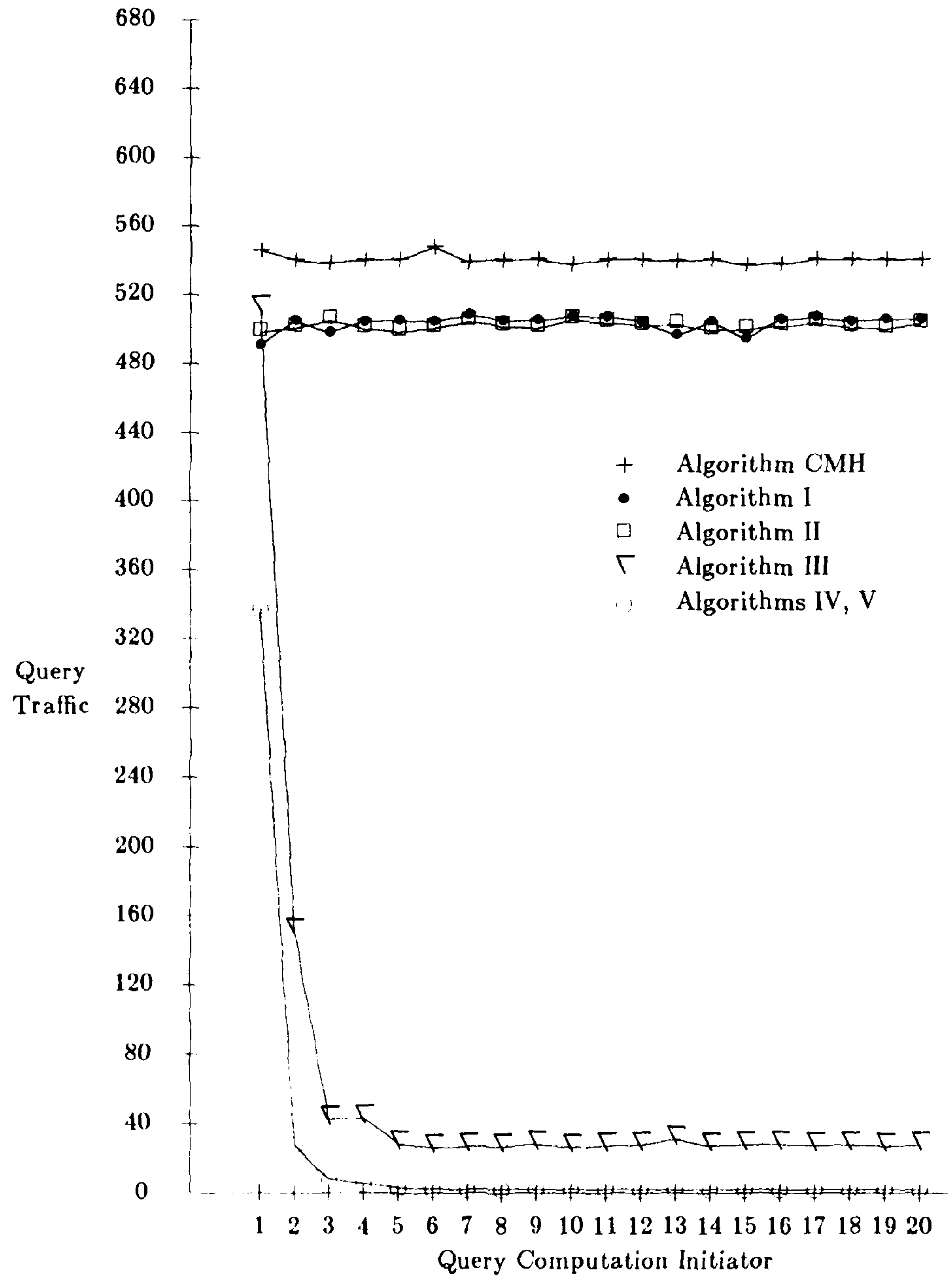


Figure 28

Volume of Query Traffic for Test Configur..tion 2

With Five Small Clusters and Medium to Large Size Dependent Sets

5.7.6. Case Study 6

The sixth case study evaluates the average number of data fields transmitted by all 20 processes during a single simulation run. We assume that all fields in the query traffic (except the bit flags used in Algorithm V) are the same length, and that each *query*, *reply*, and *cancel* requires one header field, also of that same length. We also assume that the bit flags used in Algorithm V add 1 bit for each 16 bits (2 fields) used in the query trace list. This implicitly assumes that each data field is 8 bits long. The input parameters for the four test configurations are shown in Table 8. The results in Tables 9 through 12 show that the new algorithms always require less query traffic than the CMH Algorithm.

The results from Test Configurations 1 and 2 show that the CMH Algorithm requires significantly fewer data fields to be transmitted than Algorithms I and II when deadlock exists because the CMH Algorithm uses fixed length *queries* and *replies*. Algorithms IV and V require fewer data fields to be transmitted than the CMH Algorithm in all four test configurations. The ratios of data fields transmitted to query traffic communications sent show that, when deadlock exists, most *queries* and *replies* for the new algorithms have fairly long query trace lists.

The results from Test Configurations 3 and 4 show that Algorithms I, III, IV, and V all require significantly fewer data fields to be transmitted than the CMH Algorithm when deadlock does not exist. The results from Test Configuration 4 also show that for larger sized dependent sets, Algorithm II also requires fewer data fields to be transmitted than the CMH Algorithm. The ratios show that, when deadlock does not exist, most *queries* and *replies* for the new algorithms have relatively short query trace lists. For Algorithms

III, IV, and V, the priority system ensures that most query computations are suspended due to higher priority query computations, and hence their query trace lists are kept short.

Table 8
Input Parameters for Case Study 6

Parameter	Test Config. 1 Deadlock Five Small Clusters Small to Medium Dependent Sets	Test Config. 2 Deadlock Five Small Clusters Medium to Large Dependent Sets	Test Config. 3 No Deadlock Five Small Clusters Small to Medium Dependent Sets	Test Config. 4 No Deadlock Five Small Clusters Medium to Large Dependent Sets
Processes in cluster 1	4	4	4	4
Processes in cluster 2	3	3	3	3
Processes in cluster 3	4	4	4	4
Processes in cluster 4	5	5	5	5
Processes in cluster 5	4	4	4	4
Intra-, Inter-cluster delay times (in hops)	2, 5	2, 5	2, 5	2, 5
Mean time a process executes until next event	20	20	20	20
Probability a process sends a message for its next event (constraints)	50 (0,20)	50 (0,20)	50 (3,20)	50 (3,20)
Number of processes in dependent set from same cluster	1-2	2-3	1-2	2-3
Number of processes in dependent set from other clusters	1-7	8-15	1-7	8-15
Times T_1 and T_2 a process waits before processing queries	500,450	500,450	500,450	500,450

Table 9
Average Data Field Requirements with Deadlock, Test Configuration 1 Using Five Small Clusters and Small to Medium Size Dependent Sets

	Alg CMH	Alg I	Alg II	Alg III	Alg IV	Alg V
data fields used	14932	55632	101086	16564	7542	7994
query traffic comms	3733	2793	2857	447	226	226
ratio	4.0	19.9	35.4	37.1	33.4	35.4

Table 10
Average Data Field Requirements with Deadlock, Test Configuration 2 Using Five Small Clusters and Medium to Large Size Dependent Sets

	Alg CMH	Alg I	Alg II	Alg III	Alg IV	Alg V
data fields used	43188	212887	382690	46551	14281	15111
query traffic comms	10797	10070	10017	1182	415	415
ratio	4.0	21.1	38.2	39.4	34.4	36.4

Table 11
Average Data Field Requirements without Deadlock, Test Configuration 3 Using Five Small Clusters and Small to Medium Size Dependent Sets

	Alg CMH	Alg I	Alg II	Alg III	Alg IV	Alg V
data fields used	3784	710	5742	210	208	212
query traffic comms	946	112	437	33	32	32
ratio	4.0	6.3	13.2	6.5	6.4	6.6

Table 12
Average Data Field Requirements without Deadlock, Test Configuration 4 Using Five Small Clusters and Medium to Large Size Dependent Sets

	Alg CMH	Alg I	Alg II	Alg III	Alg IV	Alg V
data fields used	892	23	176	9	9	9
query traffic comms	223	4	16	2	2	2
ratio	4.0	5.5	10.7	6.0	6.0	6.0

5.7.7. Case Study 7

The seventh case study compares the volume of query traffic as a function of the time-weighted average number of dependency relationships defined by equation (1). The input parameters for the three test configurations are shown in Table 13. During a simulation run, the size of the dependent sets was held constant. Each test configuration was simulated 50 times for each size of dependent set. Results of simulations with a dependent set size of 4 are not shown for Test Configurations 2 and 3 due to difficulty in obtaining deadlock-free simulations without changing other input parameters such as the probability a process sends a message for its next event or the minimum number of processes to be kept executing. The smaller values of T_1 and T_2 in Test Configuration 3 show more clearly that all the algorithms are very sensitive both to the number of dependency relationships and the values of T_1 and T_2. Figure 29 shows that as the dependent set size increases, the average number of idle processes decreases. This happens because the probability that an idle process will accept a new message increases as the dependent set size increases. We tried to minimize this effect by requiring the following preference scheme be used when any process B sends any message. The destination process A is chosen in the following order of decreasing preference:

(1) A is an idle process which has B in its dependent set.

(2) A is an executing process.

(3) A is an idle process which does not have B in its dependent set.

Figure 30 shows that as the dependent set size increases, the average number of dependency relationships increases linearly.

The results of the comparison between the volume of query traffic and the number of dependency relationships are shown in Figures 31 through 45. The data are the same for Algorithms IV and V, so they are plotted together.

Table 13
Input Parameters for Case Study 7

Parameter	Test Config. 1 Deadlock One Large Cluster	Test Config. 2 No Deadlock One Large Cluster	Test Config. 3 No Deadlock One Large Cluster
Number of processes in cluster 1	20	20	20
Intra–cluster delay times	2	2	2
Mean time a process executes until next event	20	20	20
Probability a process sends a message for its next event (constraints)	50 (0,20)	50 (3,20)	50 (3,20)
Number of processes in dependent set	4,6,8,10,12 14,16,18	6,8,10,12 14,16,18	6,8,10,12 14,16,18
Times T_1 and T_2 a process waits before processing query computations	500, 450	500, 450	100, 90

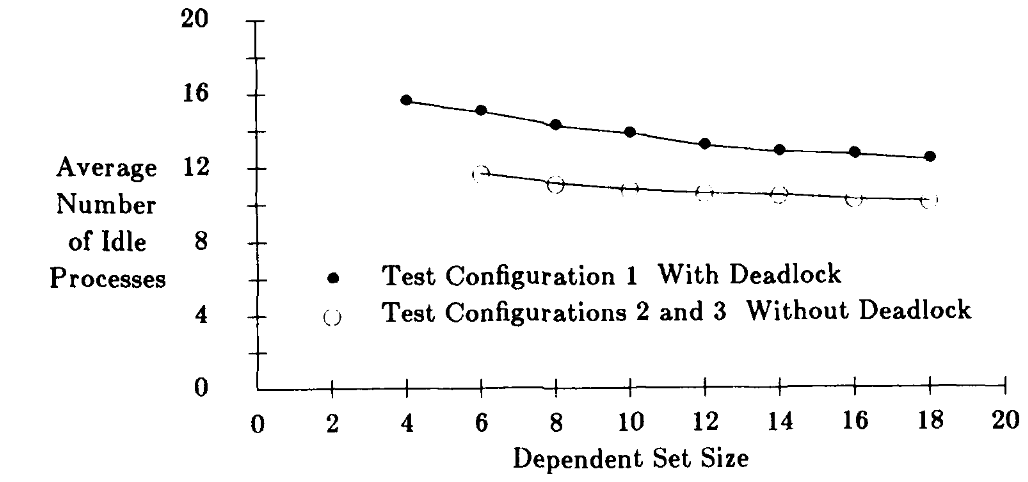


Figure 29
Average Number of Idle Processes

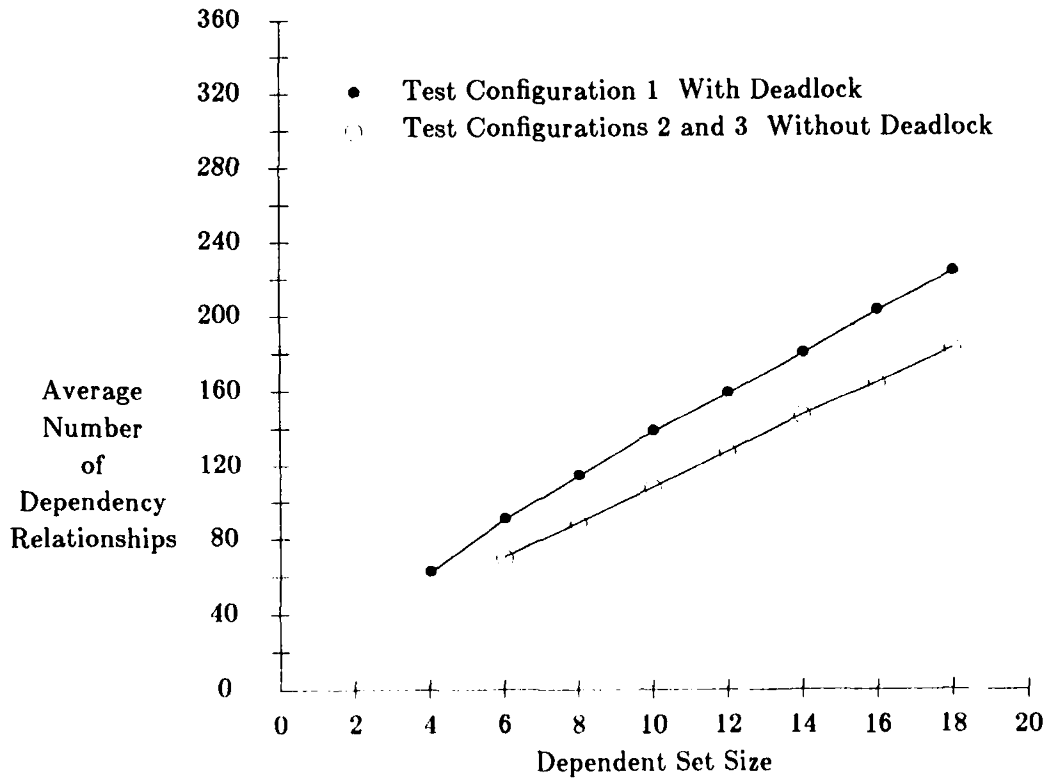


Figure 30
Average Number of Dependency Relationships

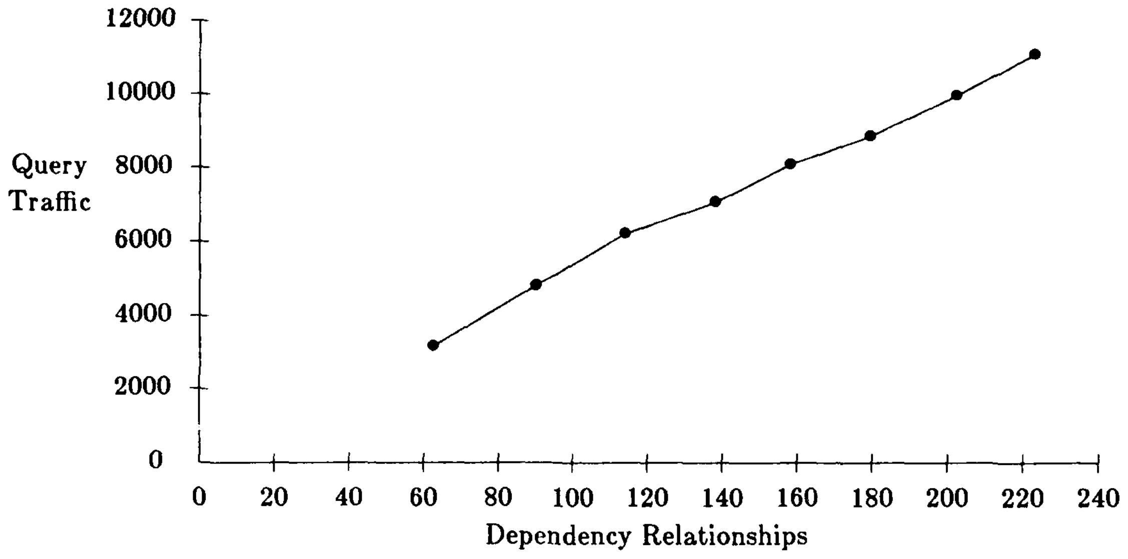


Figure 31

Query Traffic for Algorithm CMH, Test Configuration 1 With Deadlock

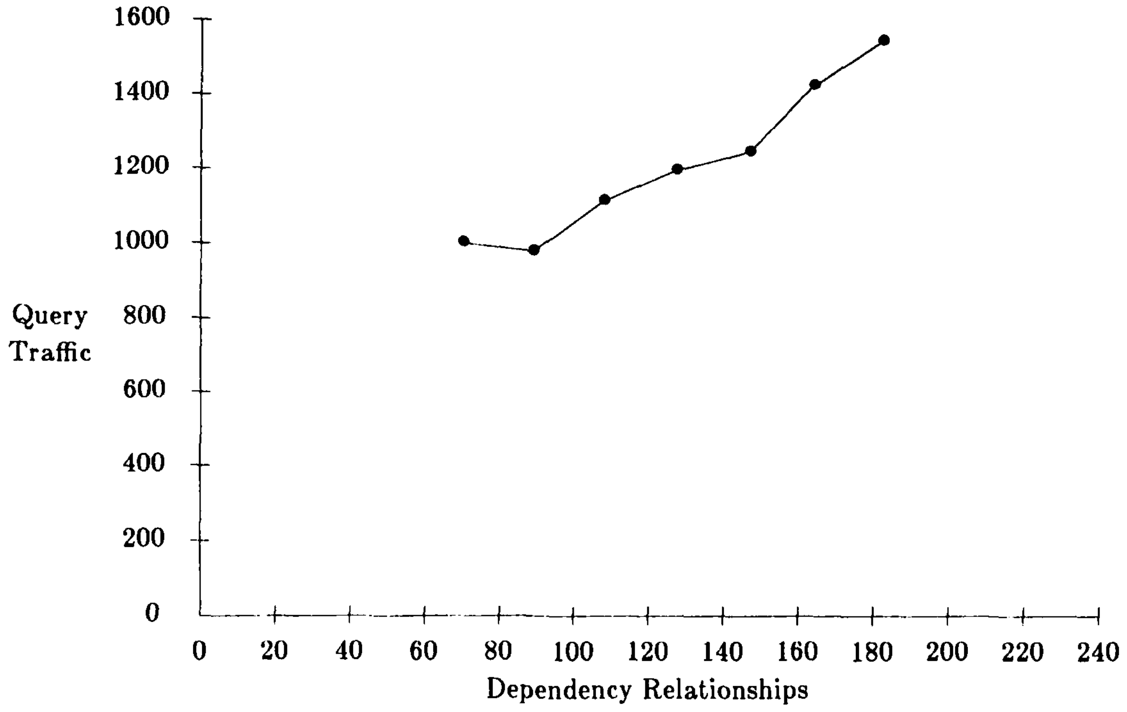


Figure 32

Query Traffic for Algorithm CMH, Test Configuration 2 Without Deadlock

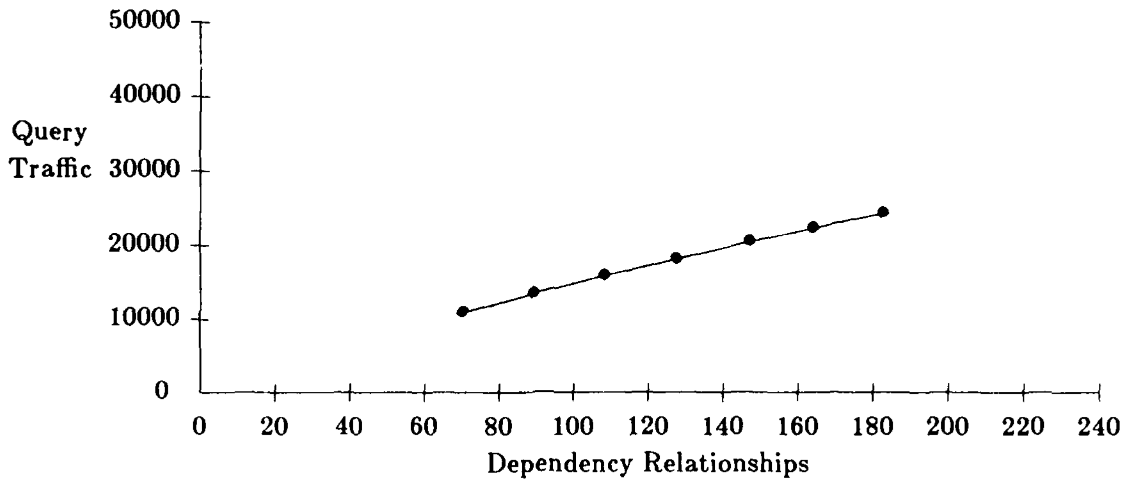


Figure 33
Query Traffic for Algorithm CMH, Test Configuration 3 Without Deadlock

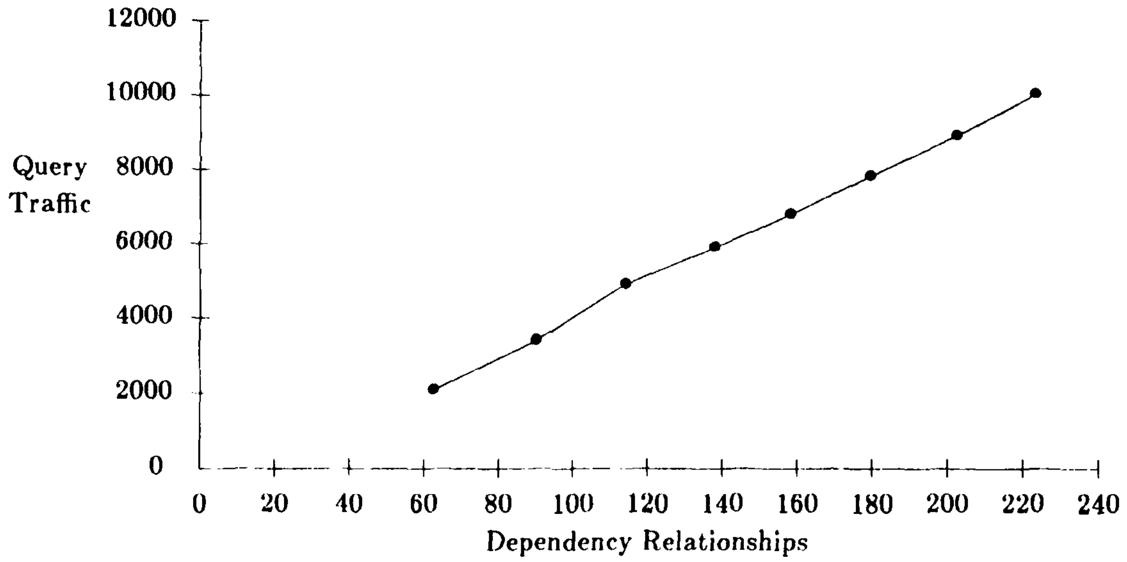


Figure 34
Query Traffic for Algorithm I, Test Configuration 1 With Deadlock

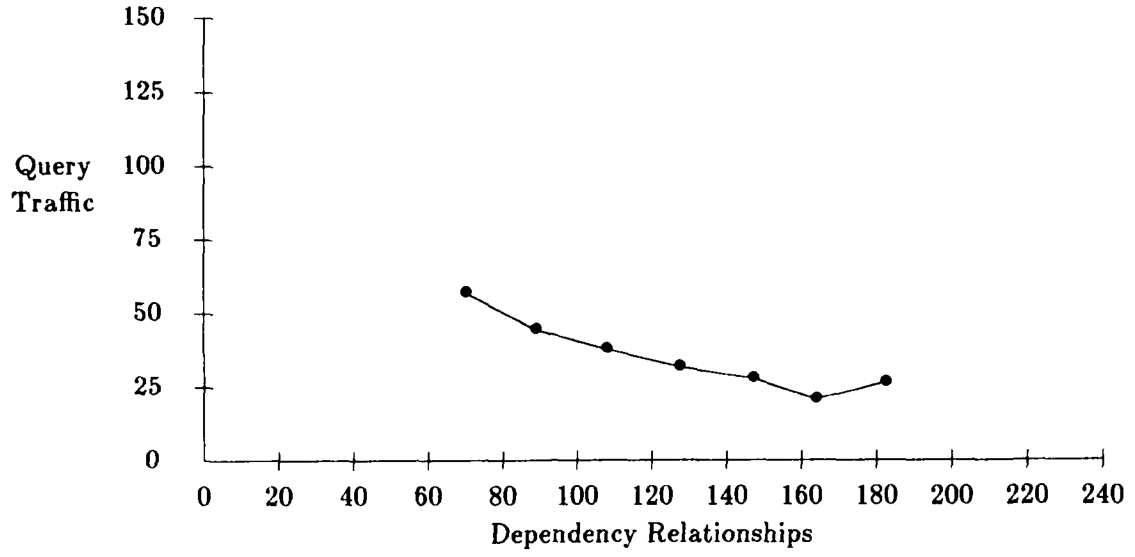


Figure 35

Query Traffic for Algorithm I, Test Configuration 2 Without Deadlock

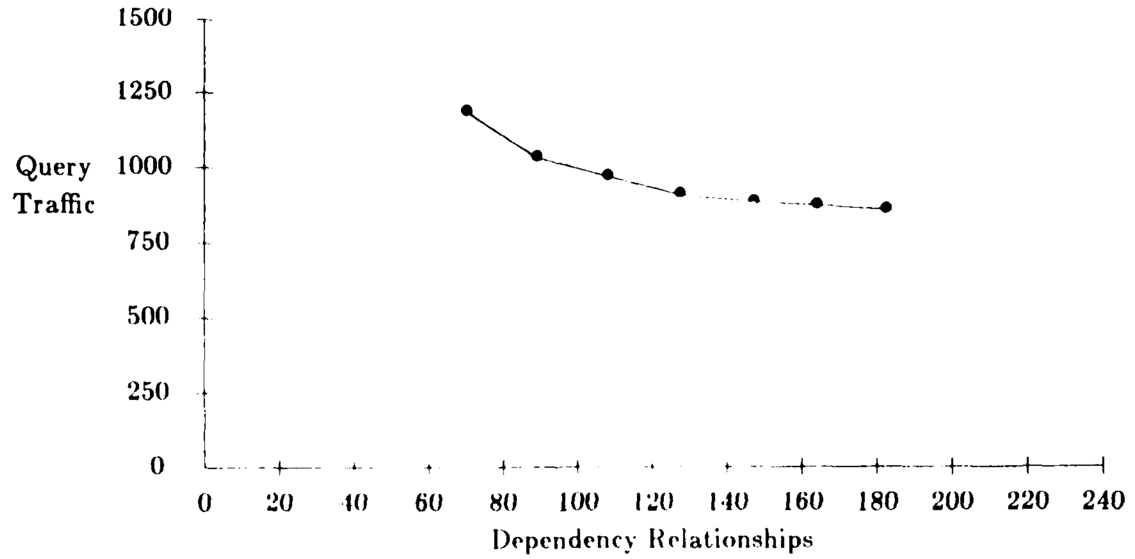


Figure 36

Query Traffic for Algorithm I, Test Configuration 3 Without Deadlock

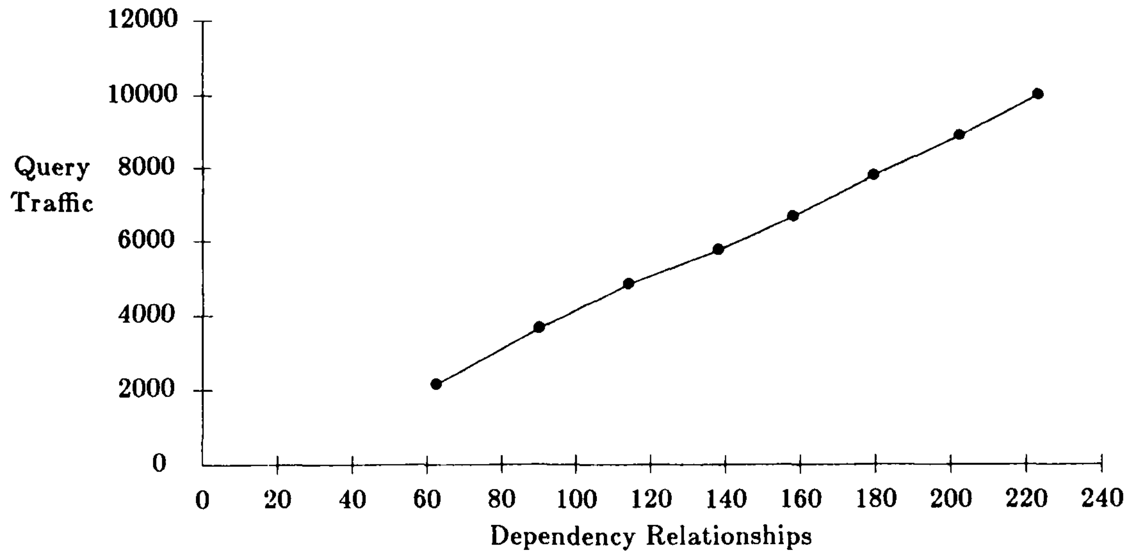


Figure 37
Query Traffic for Algorithm II, Test Configuration 1 With Deadlock

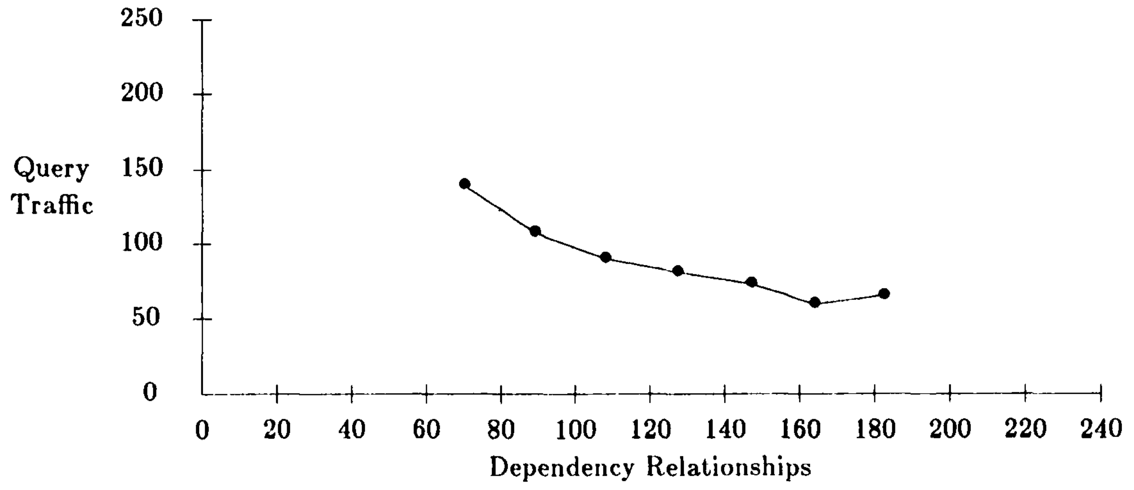


Figure 38
Query Traffic for Algorithm II, Test Configuration 2 Without Deadlock

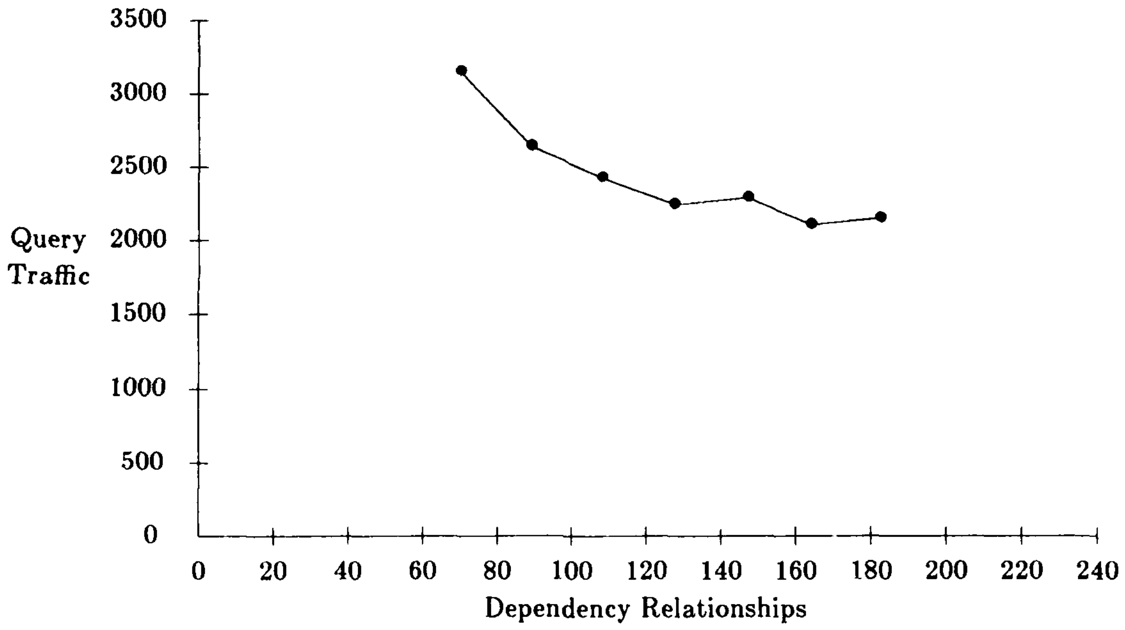


Figure 39

Query Traffic for Algorithm II, Test Configuration 3 Without Deadlock

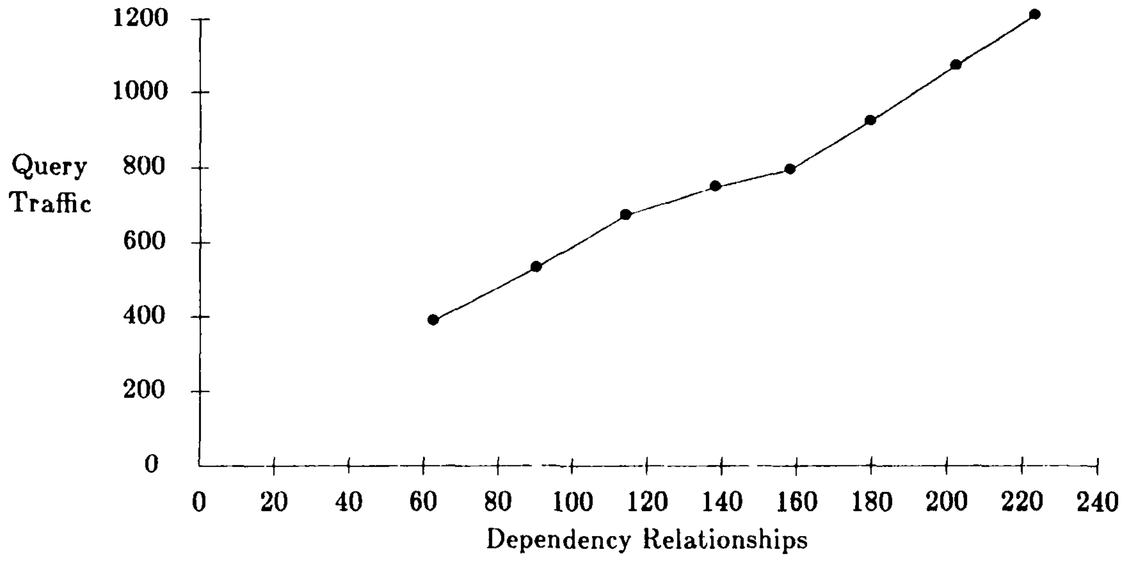


Figure 40

Query Traffic for Algorithm III, Test Configuration 1 With Deadlock

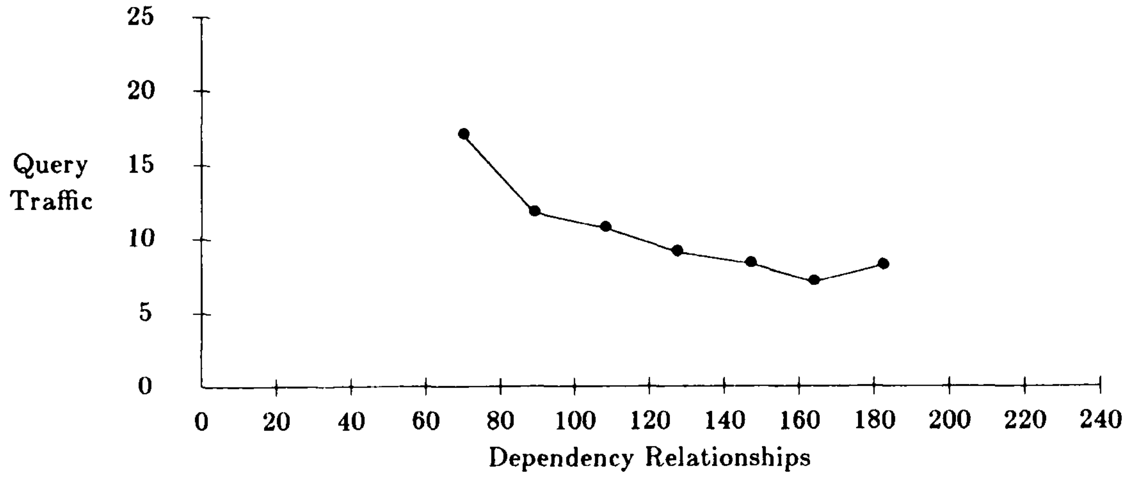


Figure 41
Query Traffic for Algorithm III, Test Configuration 2 Without Deadlock

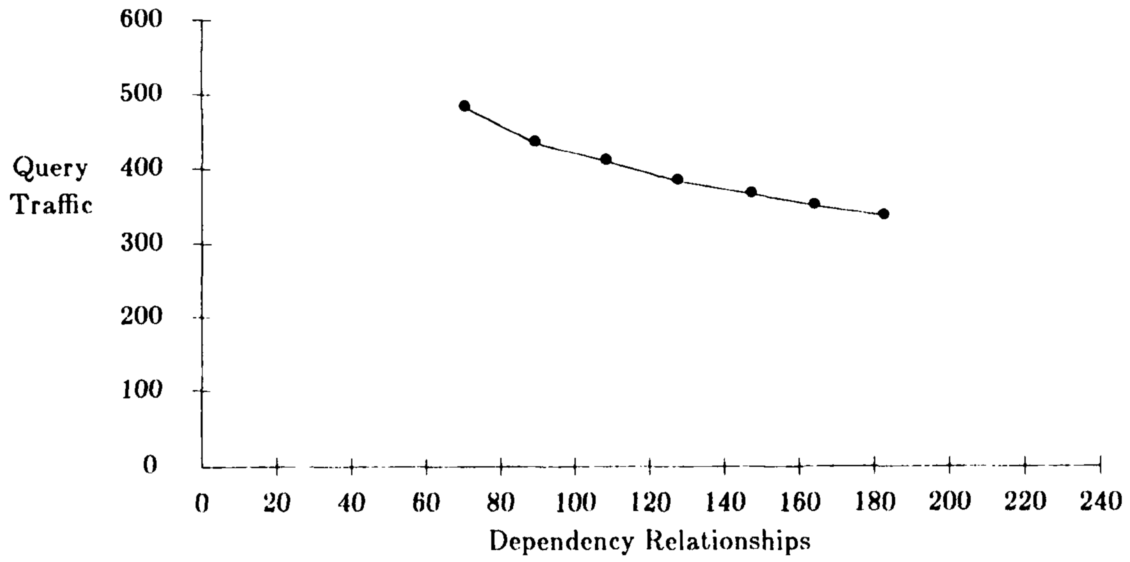


Figure 42
Query Traffic for Algorithm III, Test Configuration 3 Without Deadlock

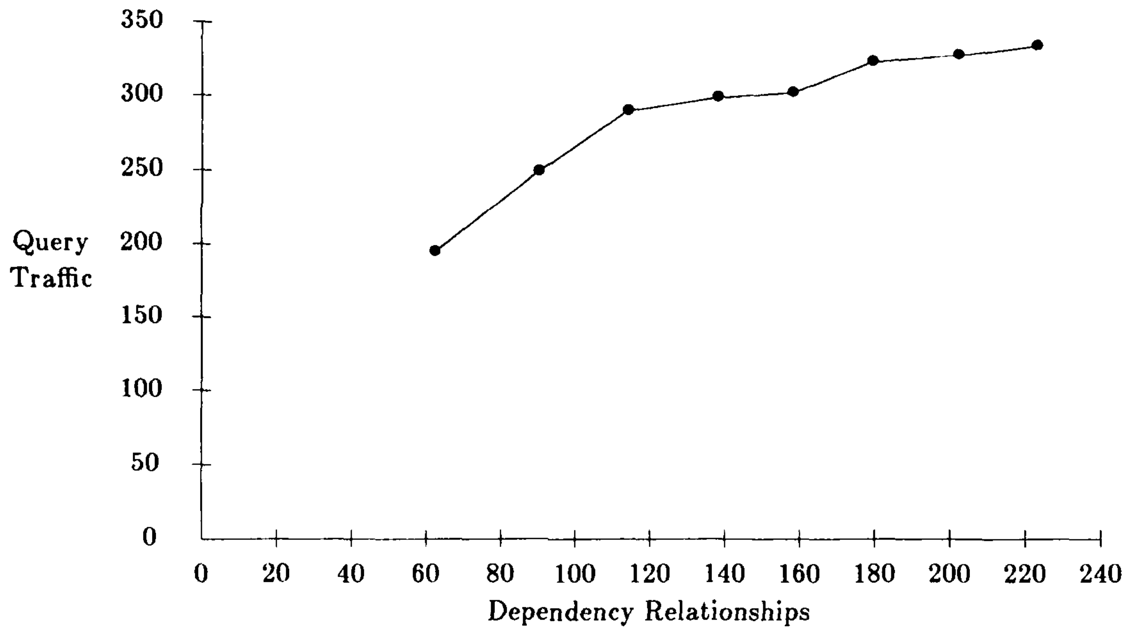


Figure 43

Query Traffic for Algorithms IV and V, Test Configuration 1 With Deadlock

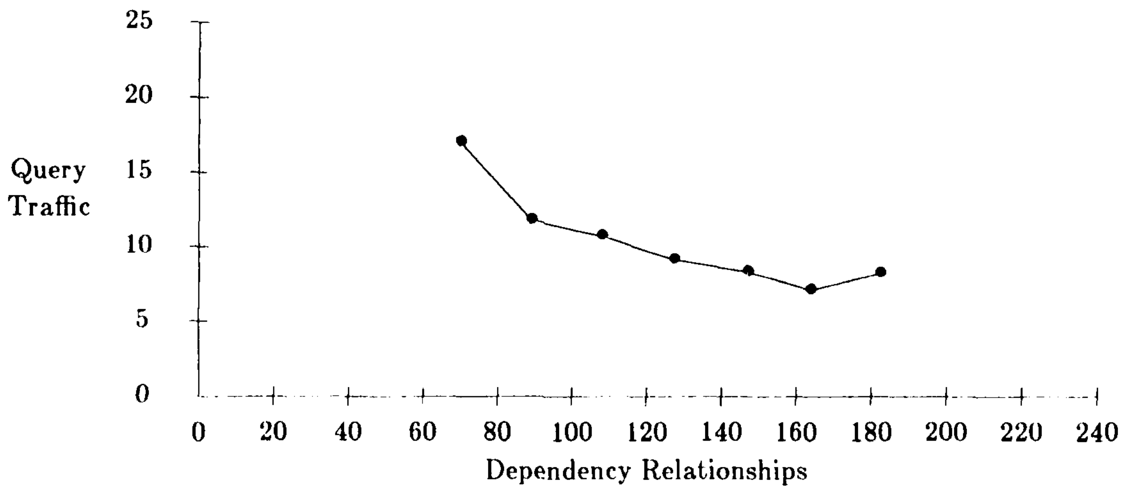


Figure 44

Query Traffic for Algorithms IV and V, Test Configuration 2 Without Deadlock

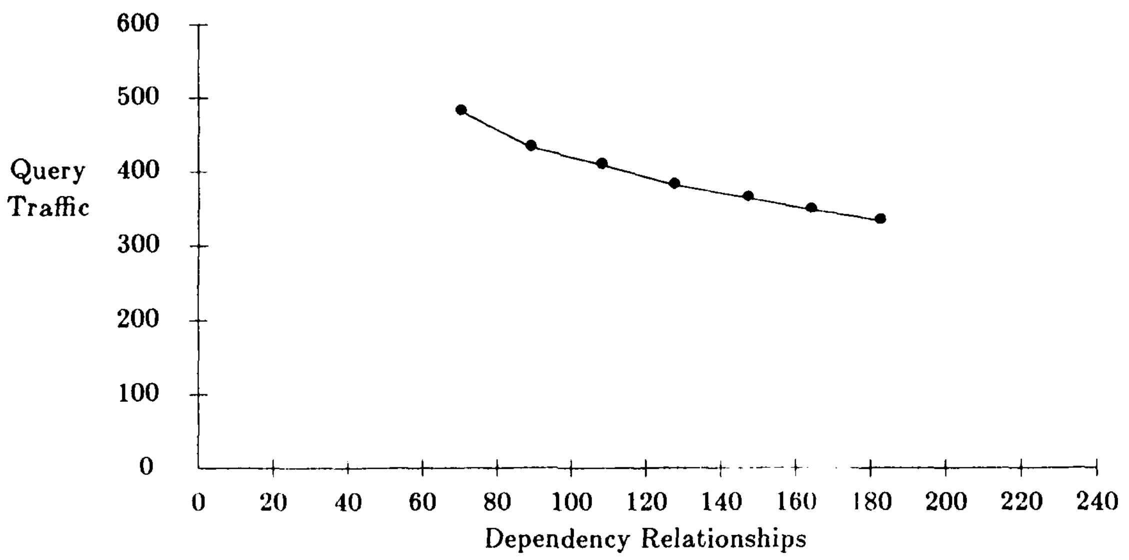


Figure 45
Query Traffic for Algorithms IV and V, Test Configuration 3 Without Deadlock

To detect deadlock, each query computation in the CMH Algorithm requires one *query* and *one reply for each dependency relationship.* As expected, Figure 31 shows that the volume of query traffic increases linearly with respect to the average number of dependency relationships when the number of processes is held constant. Figures 32 and 33 show similar results when deadlock does not exist. Algorithms I and II also exhibit similar behavior when deadlock exists. When deadlock does not exist, the volume of query traffic for all the new algorithms decreases as the number of dependency relationships increases. This can also be observed in the results from Case Study 2. This happens because the average number of idle processes also decreases as shown in Figure 29. As the number of executing processes increases, the probability that a query computation will encounter one of these executing processes also increases. With the depth-first approach of the new algorithms, when any executing process is encountered, the entire query computation is either halted or suspended. This is not the case for the

CMH Algorithm which does not halt or suspend query computations.

Figure 43 seems to suggest that, when deadlock exists for Algorithms IV and V, the volume of query traffic grows sub-linearly with the number of dependency relationships when the number of processes in the system is held constant. Let $A \longrightarrow B$ denote the relationship of B being in the dependent set of A. In this simulation, as the number of dependency relationships increased, the number of dependency relationships $A \longrightarrow B$ for which no message was sent from B to A during the course of the simulation also increased. The curve may be straightened by running the simulation until, for every pair of processes (A,B), A has sent at least one message to B.

CHAPTER 6.

Conclusions and Recommendations

6.1. Introduction

Five algorithms which use variable-length *queries* and *replies* to detect deadlock have been presented. Instead of using timeout to indicate an absence of deadlock, these algorithms use explicit messages called *informs* to convey the absence of deadlock to the initiator of a query computation. Algorithm I detects a deadlock if it existed when the query computation was initiated. The other algorithms also detect deadlock if deadlock conditions develop after the query computation is initiated. Proofs of correctness have been provided for the algorithms. A simulation study was conducted to compare their performance with that of the CMH Algorithm.

6.2. Conclusions

For distributed deadlock detection among communicating processes, the use of *informs*, *cancels*, and the depth-first approach has been shown to significantly reduce both the volume of query computation traffic and the number of data fields transmitted. The use of *informs* to notify processes directly when they are not deadlocked allows processes to discover they are not deadlocked much sooner. Our simulation results show that the average length of the query trace list was found to be very short when deadlock does not exist. In general, only a few *queries* were sent for a query computation, followed by an *inform*. Thus, the initiator learned it was not deadlocked only a few message delays after initiating the query computation. The CMH Algorithm always requires a wait at least as long as $2N$ message delays. The depth-first algorithms are also less sensitive than the CMH Algorithm to the amount of time a process waits before initiating a query computation. By adding a priority scheme, using information from previous query computations, and using extra bit flags in the query trace list, a process can learn the identity of all minimal deadlocked sets in its reachable set with a minimum of query traffic.

For the CMH Algorithm and Algorithms I and II, query computations operate independently of each other. Each process, after becoming idle, initiates a query computation. Each query computation uses one *query* and one *reply* for each dependency relationship between processes in the reachable set of the initiator. Thus, for a system with N processes, the CMH Algorithm and Algorithms I and II may require $O(N^2)$ *queries* and *replies* for each query computation, or a total of $O(N^3)$ *queries* and *replies*. Algorithms III, IV and V, however, only require $O(N^2)$ *queries* and *replies* for the highest priority query computation. Our simulation results show that for Algorithm III, the other lower priority query computations only require $O(N)$ *queries* and *replies*. For Algorithms IV and V, the other lower priority query computations only require a constant number of *queries* and *replies*. Thus, Algorithms III, IV, and V require a total of $O(N^2)$ *queries* and *replies*. This assumes, of course, that each process has a unique priority.

With the CMH Algorithm, it is assumed that any process which detects deadlock simply breaks the deadlock in some manner. However, some processes which detect deadlock may not be in a minimal deadlocked set. For them to try to break a deadlock is futile. Also, more than one process in a minimal deadlocked set may try to break the deadlock. This may be wasteful, especially if they are processes which must undo a relatively large amount of work. Algorithm V identifies all the minimal deadlocked sets in the reachable set of the initiator. Therefore, only processes in minimal deadlocked sets will participate in a scheme to break the deadlock. Since the minimal deadlocked set is fully identified, an efficient method of breaking the deadlock is more likely to be found.

The CMH Algorithm requires a process which sends out M *queries* to receive M *replies* before sending a *reply* to its engager. Duplicate *replies* render this approach impractical and prone to false deadlock detection, although Chandy, Misra, and Haas circumvent this by assuming a completely reliable message delivery system. By requiring the new algorithms to match a *reply* with the corresponding *query*, the new algorithms are not affected by the problem of duplicate *queries* and *replies*. Lost query traffic and network partitioning remain a problem, of course.

The CMH Algorithm and Algorithm I both use storage proportional to N for each process, whereas Algorithms II through V use storage proportional to N^2 for each process. This extra storage is required so that processes can suspend and later resume query computations. This is a requirement of the priority scheme first introduced in Algorithm III.

6.3. Recommendations

Three modifications should be incorporated into the CMH Algorithm. First, the *inform* should be used. This modification would require that executing processes be able to be interrupted so that they can process query computations. This would allow a process to discover it is not deadlocked quickly instead of requiring it to assume so after a lengthy wait. Second, a process which detects deadlock should initiate a second phase of deadlock detection which involves notifying all members of its reachable set. This also would be a breadth-first approach, with the notification message that deadlock was detected being sent first to processes in the dependent set of the initiator, and then having them pass the information along to their dependent sets, and so forth. This may prevent some of the processes from initiating their own query computations. It would also be a first step toward breaking deadlocks more efficiently. Third, when a process receives an engaging query, it does not need to query its engager if its engager is a member of its dependent set.

The algorithms proposed in this thesis should also incorporate a provision for having the initiator notify processes in its reachable set that they form a deadlocked set. For systems with a reliable broadcast capability, this could be a one–step process, since the query trace list identifies all such processes.

The algorithms proposed in this thesis need to be sensitive to the network structure when processes are clustered. That is, if inter–cluster delay times are significantly larger than intra–cluster delay times, then the number of inter–cluster *queries* and *replies* should be kept to an absolute minimum. Also, inter–cluster querying should be postponed as long as possible. To accomplish this, processes would have to know the identities of the processes in each cluster. A process could then query the processes in its dependent set which are in the same cluster before querying processes belonging to other clusters. Work by Goldman [Gold77], Obermarck [Ober80], and Tsai [Tsai82] may also offer suggestions in this area.

References

[AhHo74] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[BeOb81] Beeri, Catriel and Ron Obermarck. "A Resource Class Independent Deadlock Detection Algorithm." IBM Research Laboratory, San Jose, California, RJ3077, 6 March 1981.

[Brac85] Bracha, Gabriel. "Randomized Agreement Protocols and Distributed Deadlock Detection Algorithms." Technical Report TR-85-657, Department of Computer Sciences, Cornell University, Ithaca, New York, January 1985.

[BrTo84] Bracha, Gabriel and Sam Toueg. "A Distributed Algorithm for Generalized Deadlock Detection." *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* Vancouver, B. C., Canada, pp. 285-301, 27-29 August 1984.

[ChLa85] Chandy, K. Mani and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems,* vol. 3, no. 1, pp. 63-75, February 1985.

[ChMi79] Chandy, K. M. and J. Misra. "Deadlock Absence Proofs for Networks of Communicating Processes." *Information Processing Letters,* vol. 9, no. 4, pp. 185-189, 20 November 1979.

[ChMi82] Chandy, K. M. and J. Misra. "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems." *Proceedings, ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* (Ottawa, Canada, 18-20 August 1982), pp. 157-164.

[ChMi83] Chandy, K. Mani, Jayadev Misra, and Laura M. Haas. "Distributed Deadlock Detection." *ACM Transactions on Computer Systems,* vol. 1, no. 2, pp. 144 156, May 1983.

[ChMi85] Chandy, K. Mani and Jayadev Misra. "A Paradigm for Detecting Quiescent Properties in Distributed Computations." Technical Report TR-85-02, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, January 1985.

[CoEl71] Coffman, E. G., Jr., M. J. Elphick, and A. Shoshani. "System Deadlocks." *Computing Surveys,* vol. 3, no. 2, pp. 67-78, June 1971.

[CoLe82] Cohen, Shimon and Daniel Lehmann. "Dynamic Systems and Their Distributed Termination." *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Ottawa, Ontario, Canada, 18-20 August 1982, pp. 29-33.

[DiSc80] Dijkstra, Edsger W. and C. S. Scholten. "Termination Detection for Diffusing Computations." *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 29 August 1980.

[FiLy85] Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." *Journal of the Association for Computing Machinery*, vol. 32, no. 2, pp. 374–382, April 1985.

[Fran80] Francez, Nissim. "Distributed Termination." *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 1, pp. 42–55, January 1980.

[GlSh80] Gligor, Virgil D. and Susan H. Shattuck. "On Deadlock Detection in Distributed Systems." *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 435–440, September 1980.

[Gold77] Goldman, B. "Deadlock Detection in Computer Networks." Technical Report MIT-LCS-TR185, Massachusetts Institute of Technology, Cambridge, Mass., September 1977.

[GrHo81] Gray, Jim, Pete Homan, Ron Obermarck, and Hank Korth. "A Straw Man Analysis of Probability of Waiting and Deadlock." IBM Research Laboratory, San Jose, California, RJ3066, 26 February 81.

[Haas81] Haas, Laura Myers. "Two Approaches to Deadlock in Distributed Systems." Thesis, University of Texas at Austin, Austin, Texas, August 1981.

[Hoar78] Hoare, C.A.R. "Communicating Sequential Processes." *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, August 1978.

[Holt72] Holt, Richard C. "Some Deadlock Properties of Computer Systems." *Computing Surveys*, vol. 4, no. 3, pp. 179–196, September 1972.

[IsMa78] Isloor, Srreekaanth S. and T. Anthony Marsland. "An Effective On-line Deadlock Detection Technique for Distributed Data Base Management Systems." *IEEE Proceedings Compsac 78*, Chicago, Ill., pp. 283–288, November 1978.

[LaMe85] Lamport, Leslie and P. M. Melliar-Smith. "Synchronizing Clocks in the Presence of Faults." *Journal of the Association for Computing Machinery*, vol. 32, no. 1, pp. 52–78, January 1985.

[Leun83] Leung, Joseph Y-T. "Complexity of Optimal Deadlock Recovery." *Proceedings, Twenty-first Annual Allerton Conference on Communication, Control, and Computing*, 5–7 October 1983, pp. 876–885.

[MaIs80] Marsland, T. Anthony and Sreekaanth S. Isloor. "Detection of Deadlocks in Distributed Database Systems." *INFOR*, vol. 18, no. 1, pp. 1–20, February 1980.

[MeMu79] Menasce, Daniel A. and Richard R. Muntz. "Locking and Deadlock Detection in Distributed Data Bases." *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 195–202, May 1979.

[MiCh82] Misra, Jayadev and K. M. Chandy. "Termination Detection of Diffusing Computations in Communicating Sequential Processes." *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, pp. 37-43, January 1982.

[Misr83] Misra, Jayadev. "Detecting Termination of Distributed Computations Using Markers." *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Canada, 17-19 August 1983, pp. 290-294.

[MiMe84] Mitchell, Don P. and Michael J. Merritt. "A Distributed Algorithm for Deadlock Detection and Resolution." *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, 27-29 August 1984, pp. 282-284.

[Ober80] Obermarck, Ron. "Global Deadlock Detection Algorithm." IBM Research Laboratory, San Jose, California, RJ2845, 13 June 1980.

[Ober82] Obermarck, Ron. "Distributed Deadlock Detection Algorithm." *ACM Transactions on Database Systems*, vol. 7, no. 2, pp. 187-208, June 1982.

[ReKa79] Reed, David P. and Rajendra K. Kanodia. "Synchronization with Eventcounts and Sequencers." *Communications of the ACM*, vol. 22, no. 2, pp. 115-123, February 1979.

[ReNi77] Reingold, Edward M., Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[ShSi85] Sheth, Amit P., Anoop Singhal, and Ming T. Liu. "An Analysis of the Effect of Network Parameters on the Performance of Distributed Database Systems." *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1174-1184, October 1985.

[SiNa85] Sinha, Mukul K. and N. Natarajan. "A Priority Based Distributed Deadlock Detection Algorithm." *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, pp. 67-80, January 1985.

[SzSh85] Szymanski, Boleslaw, Yuan Shi, and Noah S. Prywes. "Synchronized Distributed Termination." *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1136-1140, October 1985.

[Tane81] Tanenbaum, Andrew S. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Tsai82] Tsai, Wang-Chuan. "Distributed Deadlock Detection in Distributed Database Systems." Department of Computer Science, University of Illinois, Urbana, Illinois, Report No. UIUCDCS-R-82-1089, 15 January 1982.

Appendix A.

Code for Algorithm I

In the code shown below, *QTL* represents the query trace list of the current query computation. It is assumed that the *QTL* is variable in length and is as short as possible. The following subprograms are used in the code shown below for Algorithm I.

COMBIN	Procedure **COMBIN** accepts a query trace list (*QTL*) and adds to it any processes which are in the reachable set of the deadlocked calling process.
NXTQRY	Procedure **NXTQRY** determines the next process in the dependent set which must be queried for a given query computation. It returns in its fourth argument P_S a value of zero if no more processes in the dependent set need to be queried, or else the process number of the next process to receive the query.

Following is the pseudo-code for Algorithm I.

(1) When an idle process P_r receives a message and begins executing, the following occurs:

```
$T$ := clocktime;
$STATE$ := TRUE;
for $F$ := 1 to $N$ do
begin
    $WAIT(F)$ := FALSE;
    if $TARGET(F) \neq 0$
                                (* Find all outstanding queries *)
    then begin
                                (* Send inform to query initiator *)
        if $P_r \neq P_F$
            then send inform I($P_r$, $PF$, $LATEST(F)$, $T$);
                                (* Cancel all outstanding queries *)
                                (* which created tree edges when sent *)
        if $EDGETYPE(F)$
            then send cancel C($P_r$, $TARGET(F)$, $LATEST(F)$, $P_F$);
        $TARGET(F) := 0$;
    end;
    endif
                                (* Remove all traces of queries received *)
    $ENGAGER(F) := 0$;
end;
```

(2) When a process P_r receives a *query* Q(P_k, P_r, M, P_i, ..., P_j), the following occurs:

```
$T$ := clocktime;
if ($M > LATEST(i)$)
                                (* do not process old queries *)
then begin
    $LATEST(i) := M$;
    if $STATE$
                                (* If executing, send inform to query initiator *)
    then send inform I($P_r$, $P_i$, $M$, $T$)
    else begin
        if $DEADLK$
        then begin
            $WAIT(i)$ := TRUE;
                                (* Combine the query trace list $QTL$ *)
                                (* with the reachable set of $P_r$ *)
            COMBIN($QTL$, $P_r$);
                                (* Send the reply using the *)
                                (* augmented query trace list *)
            send reply R($P_r$, $P_k$, $M$, $QTL$);
        end
        else begin
            if $P_r$ is in the query trace list
            then begin
                if $WAIT(i)$
                then send reply R($P_r$, $P_k$, $M$, $P_i$, ..., $P_r$, ..., $P_j$)
                else send inform I($P_r$, $P_i$, $M$, $T$);
            end
            else begin
                $WAIT(i)$ := TRUE;
                                (* find the next process to send a query to *)
                NXTQRY($P_r$, 0, $P_k$, $P_S$);
                                (* if all members of the dependent set have *)
                                (* been queried by $P_r$, then send a reply *)
                                (* to the process which sent the query *)
                if $P_S = 0$
                then begin
                    send reply R($P_r$, $P_k$, $M$, $P_i$, ..., $P_j$, $P_r$);
                    $ENGAGER(i) := 0$;
                end
                else begin
```

```
                                        (* else send a $query$ to the first process *)
                                        (* in the dependent set *)
                              send $query$ Q($P_r$, $P_S$, $M$, $P_i$, ..., $P_j$, $P_r$);
                              if $P_S$ is in the query trace list
                              then $EDGETYPE(i) :=$ FALSE
                              else $EDGETYPE(i) :=$ TRUE;
                              $TARGET(i) := P_S$;
                              $ENGAGER(i) := P_k$;
                         end;
                         endif;
                    end;
                    endif;
               end;
               endif;
          end;
          endif;
     end;
     endif;
```

(3) When a process P_r receives a ***reply*** R(P_k, P_r, M, P_i, ..., P_j), the following occurs:

```
     if ($ENGAGER(i) = 0$) OR ($M \neq LATEST(i)$) OR ($STATE$)
                                        (* check for process not being continuously idle, *)
                                        (* or the $reply$ not being current *)
     then null action
     else begin
          if $DEADLK$
          then begin
               COMBIN($QTL$, $P_r$);
               send $reply$ R($P_r$, $ENGAGER(i)$, $M$, $P_i$, ..., $P_j$);
               $TARGET(i) := 0$;
               $ENGAGER(i) := 0$;
          end
          else begin
                                        (* find the first member of the dependent set *)
                                        (* which has not been queried *)
               NXTQRY($P_r$, $P_k$, $ENGAGER(i)$, $P_S$);
               if $P_S = 0$
                                        (* $P_r$ has already queried all *)
                                        (* members of its dependent set *)
               then begin
                    if $P_r = P_i$
                                        (* $P_r$ is the query initiator, so declare deadlock *)
                    then DECLARE DEADLOCK for all processes $P_i$, ..., $P_j$
                    else begin
                                        (* send a $reply$ to the process *)
                                        (* which originally sent the $query$ *)
                         send $reply$ R($P_r$, $ENGAGER(i)$, $M$, $P_i$, ..., $P_j$);
                    $TARGET(i) := 0$;
                    $ENGAGER(i) := 0$;
                end;
                endif
            end
            else begin
                                        (* query next process in the dependent set *)
                                        (* which needs to be queried *)
                send query $Q(P_r, P_S, M, P_i, ..., P_j)$;
                if $P_S$ is in the query trace list
                then $EDGETYPE(i) :=$ FALSE
                else $EDGETYPE(i) :=$ TRUE;
                $TARGET(i) := P_S$;
            end;
            endif
        end;
        endif
    end;
    endif
```

(4) When a process P_r receives a *cancel* $C(P_k, P_r, M, P_i)$, the following occurs:

```
    if ($TARGET(i) \neq 0$) AND ($M = LATEST(i)$)
                                        (* if an outstanding query exists with *)
                                        (* the same initiator and matching *)
                                        (* sequence number, and the query created *)
                                        (* a tree edge when it was sent, then *)
                                        (* send the cancel on *)
    then begin
        send cancel $C(P_r, TARGET(i), M, P_i)$;
                                        (* delete references to the query being cancelled *)
        $TARGET(i) := 0$;
        $ENGAGER(i) := 0$
    end
    endif
```

(5) When a process P_r receives an *inform* $I(P_k, P_r, T)$, the following occurs:

```
                                        (* The process knows it isn't deadlocked yet, *)
                                        (* so it should use the timestamp field T *)
                                        (* in the inform to update the time used in *)
                                        (* determining when to initiate another query. *)
    if $T > LASTTIME$
    then $LASTTIME := T$
    endif
```

(6) When a process sends a message, no query traffic action is required.

(7) When an executing process P_k changes state to idle, the following occurs:

```
$STATE$ := FALSE
$LASTTIME$ := clocktime
                                        (* array $DEPENDENT$ shows which processes *)
                                        (* are in the dependent set of $P_k$ *)
```

(8) When a process P_i becomes idle for time T_1 since becoming idle or since last receiving an *inform*, the following occurs:

```
$WAIT(i)$ := TRUE;
$LATEST(i) := LATEST(i) + 1$;
                                        (* find first process in the dependent set *)
NXTQRY($P_i$, 0, 0, $P_S$);
if $P_S = 0$
                                        (* check for an empty dependent set *)
then TERMINATE THE PROCESS
else begin
                                        (* Initiate a new query computation *)
                                        (* Send a query to the first process *)
                                        (* $P_S$ in the dependent set *)
    send query Q($P_i$, $P_S$, $LATEST(i)$, $P_i$);
    $TARGET(i)$ := $P_S$;
    $ENGAGER(i)$ := $P_i$
    $EDGETYPE(i)$ := TRUE;
end;
endif
```

Appendix B.

Code for Algorithm II

In the code shown below, QTL represents the query trace list of the current query computation. It is assumed that the QTL is variable in length and is always as short as possible. The following subprograms are used in the code shown below for Algorithm II.

COMBIN	Procedure **COMBIN** accepts a query trace list (QTL) and adds to it any process and its sequence number which is in the reachable set of the deadlocked calling process.
NXTQRY	Procedure **NXTQRY** determines the next process in the dependent set which must be queried for a given query computation. It returns in its fourth argument P_S a value of zero if no more processes in the dependent set need to be queried, or else the process number of the next process to receive the query.
SEQCHK	Procedure **SEQCHK** accepts a query trace list and a process id. It returns the sequence number associated with the process id. The value zero is returned if the process is not in the query trace list or if it actually has a sequence number of zero. A process appears in a query trace list with a sequence number of zero if it was added by procedure **COMBIN** as a member of some deadlocked set.
SIG	Procedure **SIG** accepts a one-dimensional array which holds a query trace list. **SIG** returns as a query trace list the portion of the array which is non-zero (ie the significant portion).

CHP Procedure **CHP** accepts a query trace list and returns a query trace list which has been shortened so that the calling process is the last process in the query trace list.

LNG Procedure **LNG** accepts a query trace list and returns its length. Each process id/sequence number pair counts as length 2.

Following is the pseudo-code for Algorithm II.

(1) When an idle process P_r receives a message from process P_k and begins executing, the following occurs:

```
T := clocktime;
                                        (* process changes state to executing *)
STATE := TRUE;
for F := 1 to N do
    WAIT(F) := FALSE;
if TARGET(r) ≠ 0
                                        (* delete the query initiated by P_r *)
then begin
    if EDGETYPE(r)
    then begin
        send cancel C(P_r, TARGET(r), SIG (LATEST(r, *)));
                                        (* trailing zeros were truncated *)
        EDGETYPE(r) := FALSE;
    end;
    endif
    TARGET(r) := 0;
    LATEST(r, *) := 0;
    ENGAGER(r) := 0;
    TIMES(r) := 0;
end;
endif
for F := 1 to N do
begin
                                        (* for all queries remaining, send an inform *)
                                        (* to the query initiator, *)
                                        (* and cancel any outstanding queries *)
    if ENGAGER(F) ≠ 0
    then begin
        TIMES(F) := T;
        send inform I(P_r, P_F, LATEST(F [illegible]
                                        (* chop off the query trace list *)
                                        (* so P_r is the last process *)
```

```
            $LATEST(F, *)$ := **CHP** ($LATEST(F, *)$);
            if $TARGET(F) \neq 0$
            then begin
                if $EDGETYPE(F)$
                then begin
                    send *cancel* C($P_r$, $TARGET(F)$, **SIG** ($LATEST(F, *)$));
                    $EDGETYPE(F)$ := FALSE;
                end;
                endif
                $TARGET(F)$ := 0;
            end;
            endif
        end;
        endif
    end;
```

(2) When a process P_r receives a *query* Q(P_k, P_r, T, P_i, M_i, ..., P_j, M_j), the following occurs:

```
                                        (* ensure the query is valid *)
    $FLAG$ := 0;
    if $ENGAGER(i) = 0$
    then $FLAG$ := 2;
    endif
    $v$ := **LNG** ($QTL$);
    $vv$ := **LNG** (**SIG** ($LATEST(i, *)$));
    $l$ := 0;
    while ($FLAG = 0$) AND ($l < v$) AND ($l < vv$) do
    begin
        $l$ := $l$+2;
        if $QTL(l) > LATEST(i, l)$
        then $FLAG$ := 1
        else if $QTL(l) < LATEST(i, l)$
            then $FLAG$ := $-1$;
            endif
        endif
    end;
    if $FLAG \geq 0$
    then begin
        if $STATE$
        then begin
            if $P_r$ is already in the $QTL$
            then discard the *query*
            else begin
                                        (* store a copy of the query *)
                                        (* with $P_r$ added to the query trace list *)
                $LATEST(i, *)$ := $QTL$;
                $LATEST(i, v+1)$ := $P_r$;
            $LATEST(i, v+2) := 1$;
            $ENGAGER(i) := P_k$;
            $T$ := clocktime;
            $TIMES(i) := T$;
                                (* since $P_r$ is executing, send an inform *)
                                (* to the query computation initiator *)
            send inform I($P_r$, $P_i$, $LATEST(i,2)$, $T$);
        end;
        endif
    end
    else begin
        if $DEADLK$
        then begin
                                (* if $P_r$ is deadlocked, combine the reachable set *)
                                (* with the query trace list and send a reply *)
            if $P_r$ is not in the $QTL$
            then COMBIN($QTL$, $P_r$);
            endif
            send reply R($P_r$, $P_k$, $T$, $QTL$);
        end
        else begin
                                (* if $P_r$ is idle, find the first process in the *)
                                (* dependent set which is not in the $QTL$ *)
            if $P_r$ is in the $QTL$
            then begin
                if ($WAIT(i)$) OR (SEQCHK($QTL$, $P_r$) = 0)
                then send reply R($P_r$, $P_k$, $T$, $QTL$);
                endif
            end
            else begin
                $WAIT(i)$ := TRUE;
                $QTL(v+1) := P_r$;
                $QTL(v+2) := 1$;
                NXTQRY($P_r$, 0, $P_k$, $P_S$);
                if $P_S$ = 0
                                (* if every process in the dependent set *)
                                (* does not need to be queried, then *)
                                (* send a reply to $P_k$ *)
                then send reply R($P_r$, $P_k$, $T$, $QTL$)
                else begin
                                (* otherwise send a query to the *)
                                (* first process in the dependent *)
                                (* set which needs to be queried *)
                    send query Q($P_r$, $P_S$, $T$, $QTL$);
                    $TARGET(i) := P_S$;
                    $ENGAGER(i) := P_k$;
                    $TIMES(i) := T$;
                    $LATEST(i, *) := QTL$;
                    if $P_S$ is in the $QTL$
                    then $EDGETYPE(i)$ := FALSE
                    else $EDGETYPE(i)$ := TRUE;
                    endif
                end;
                endif
            end;
            endif
        end;
        endif
    end;
    endif
end;
endif
```

(3) When a process P_r receives a *reply* R(P_k, P_r, T, P_i, M_i, ..., P_j, M_j), the following occurs:

```
if $STATE$ OR ($TARGET(i) = 0$)
                                        (* ensure the reply is valid *)
then discard the reply
else begin
    $F := 1$;
    $FLAG := 0$;
    while ($FLAG = 0$ AND $LATEST(i, F) \neq 0$ AND $F < 2*N$) do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG := 1$;
        endif
        $F := F+1$;
    end;
    if $FLAG = 1$
    then discard the reply
    else begin
                                        (* at this point, the reply is valid *)
        $TARGET(i) := 0$;
        NXTQRY($P_r$, $P_k$, $ENGAGER(i)$, $P_S$);
                                        (* find the first process in the dependent *)
                                        (* set which needs to be queried *)
        if $P_S = 0$
                                        (* if every process has been queried *)
        then begin
            if $P_r = P_i$
                                        (* if $P_r$ is the query initiator, declare deadlock *)
            then begin
                DECLARE DEADLOCK for all processes in $QTL$
                $ENGAGER(r) := 0$;
```

```
                    for $F$ := 1 to $N$ do
                    begin
                        if ($ENGAGER(F) \neq 0$) AND ($r \neq F$)
                        then begin
                            $QTL$ := $LATEST(F, *)$;
                            $LATEST(F, *)$ := 0;
                            **CHP** ($QTL$);
                            if $EDGETYPE(F)$ AND ($TARGET(F) \neq 0$)
                            then begin
                                send *cancel* C($P_r$, $TARGET(F)$, $QTL$);
                                $EDGETYPE(F)$ := FALSE;
                            end;
                            endif
                            **COMBIN** ($QTL$, $P_r$);
                            send *reply* R($P_r$, $ENGAGER(F)$, $TIMES(F)$, $QTL$);
                            $ENGAGER(F)$ := 0;
                            $TARGET(F)$ := 0;
                            $TIMES(F)$ := 0;
                        end;
                        endif
                    end;
                end
                else begin
                                (* otherwise send a *reply* to the process which *)
                                (* initially sent the *query* to $P_r$ *)
                    send *reply* R($P_r$, $ENGAGER(i)$, $T$, $QTL$);
                    $ENGAGER(i)$ := 0;
                    $TARGET(i)$ := 0;
                    $TIMES(i)$ := 0;
                    $LATEST(i, *)$ := 0;
                end;
                endif
            end
            else begin
                                (* if more processes need to be queried *)
                                (* send a *query* to the first process in the *)
                                (* dependent set which needs to be queried *)
                send *query* Q($P_r$, $P_S$, $T$, $QTL$);
                $TARGET(i)$ := $P_S$;
                $TIMES(i)$ := $T$;
                if $P_S$ is already in the $QTL$
                then $EDGETYPE(i)$ := FALSE
                else $EDGETYPE(i)$ := TRUE;
                endif
            end;
            endif
        end;
        endif
```

```
end;
endif
```

(4) When a process P_r receives a *cancel* $C(P_k, P_r, P_i, M_i, ..., P_j, M_j)$, the following occurs:

```
if $ENGAGER(i) = 0$
                                        (* ensure the cancel is valid *)
then discard the cancel
else begin
    $v :=$ LNG $(QTL)$;
    $F := 1$;
    $FLAG := 0$;
    while $(FLAG = 0$ AND $F \le v)$ do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG := 1$;
        endif
        $F := F+1$;
    end;
    if $FLAG = 1$
    then discard the cancel
    else begin
                                        (* if the cancel is valid and $P_r$ has an *)
                                        (* outstanding query for the same query *)
                                        (* initiator, send the cancel to the *)
                                        (* process that $P_r$ sent the query to *)
        if $TARGET(i) \neq 0$
        then begin
            if $EDGETYPE(i)$
            then begin
                send cancel $C(P_r, TARGET(i), QTL)$;
                $EDGETYPE(i) :=$ FALSE;
            end;
            endif
            $TARGET(i) := 0$;
        end;
        endif
        $ENGAGER(i) := 0$;
        $TIMES(i) := 0$;
        $LATEST(i, *) := 0$;
    end;
    endif
end;
endif
```

(5) When a process P_r receives an *inform* $I(P_k, P_r, M, T)$, the following occurs:

```
if ($M$ = $LATEST(r, 2)$)
then if $T > LASTTIME$
    then $LASTTIME := T$;
    endif
endif
```

(6) When a process P_k sends a message to process P_r, no query computation action is required.

(7) When an executing process P_k changes state to idle, the following occurs:

```
                                        (* at this point, the array DEPENDENT *)
                                        (* shows which processes are in the *)
                                        (* dependent set of $P_k$ *)
                                        (* $P_k$ changes state to idle *)
$STATE :=$ FALSE;
$LASTTIME :=$ clocktime;
$IDLE :=$ clocktime;
                                        (* take care of all the queries being held *)
                                        (* process them as if they had just been received *)
for $G := 1$ to $N$ do
begin
    if $ENGAGER(G) \neq 0$
    then begin
        $v :=$ LNG $(LATEST(G, *))$;
        $LATEST(G, v) := LATEST(G, v)+1$;
        $T := TIMES(G)$;
        $WAIT(G) :=$ TRUE;
        NXTQRY$(P_k, 0, ENGAGER(G), P_S)$;
        if $P_S = 0$
        then begin
            send reply R$(P_k, ENGAGER(G), T,$ SIG $(LATEST(G, *)))$;
            $ENGAGER(G) := 0$;
            $LATEST(G, *) := 0$;
            $TARGET(G) := 0$;
            $TIMES(G) := 0$;
        end
        else begin
            send query Q$(P_k, P_S, T,$ SIG $(LATEST(G, *)))$;
            $TARGET(G) := P_S$;
            if $P_S$ is already in the $QTL$
            then $EDGETYPE(G) :=$ FALSE
            else $EDGETYPE(G) :=$ TRUE;
            endif
        end;
        endif
    end;
    endif
```

```
end;
```

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, the following occurs:

```
                              (* initiate a new query computation *)
$WAIT(i)$ := TRUE;
$NEWQRYNUM := NEWQRYNUM + 1$;
$LATEST(i, 1) := i$;
$LATEST(i, 2) := NEWQRYNUM$;
NXTQRY($P_i$, 0, 0, $P_S$;
if $P_S = 0$
                              (* by assumption, processes are not terminated *)
then TERMINATE THE PROCESS
else begin
    send query Q($P_i$, $P_S$, $LASTTIME$, SIG ($LATEST(i, *)$));
    $TARGET(i) := P_S$;
    $ENGAGER(i) := P_i$;
    $TIMES(i) := LASTTIME$;
    $EDGETYPE(i)$ := TRUE;
end;
endif
```

Appendix C.

Code for Algorithm III

In the code shown below, QTL represents the query trace list of the current query computation. It is assumed that the QTL is variable in length and is always as short as possible. The following subprograms are used in the code shown below for Algorithm III.

COMBIN	Procedure **COMBIN** accepts a query trace list (QTL) and adds to it any process and its sequence number which is in the reachable set of the deadlocked calling process.
NXTQRY	Procedure **NXTQRY** determines the next process in the dependent set which must be queried for a given query computation. It returns in its fourth argument P_S a value of zero if no more processes in the dependent set need to be queried, or else the process number of the next process to receive the query.
SEQCHK	Procedure **SEQCHK** accepts a query trace list and a process id. It returns the sequence number associated with the process id. The value zero is returned if the process is not in the query trace list or if it actually has a sequence number of zero. A process appears in a query trace list with a sequence number of zero if it was added by procedure **COMBIN** as a member of some deadlocked set.
URGENT	Function **URGENT** determines the urgency of a query based on its priority and the number of replies and cancels received by the process holding the query since the query was received.

SIG Procedure **SIG** accepts a one-dimensional array which holds a query trace list. **SIG** returns as a query trace list the portion of the array which is non-zero (ie the significant portion).

CHP Procedure **CHP** accepts a query trace list and returns a query trace list which has been shortened so that the calling process is the last process in the query trace list.

LNG Procedure **LNG** accepts a query trace list and returns its length. Each process id/sequence number pair counts as length 2.

Following is the pseudo-code for Algorithm III.

(1) When an idle process P_r receives a message from process P_k and begins executing, the following occurs:

```
$T$ := clocktime;
                                   (* process changes state to executing *)
$STATE$ := TRUE;
for $F$ := 1 to $N$ do
begin
    $WAIT(F)$ := FALSE;
    $STARVE(F)$ := 0;
end;
if $ENGAGER(r) \neq 0$
then begin
    if $TARGET(r) \neq 0$
                                   (* delete the query initiated by $P_r$ *)
    then begin
        if $EDGETYPE(r)$
        then begin
            send *cancel* C($P_r$, $TARGET(r)$, SIG ($LATEST(r, *)$));
                                   (* trailing zeros were truncated *)
            $EDGETYPE(r)$ := FALSE;
        end;
        endif
        $TARGET(r)$ := 0;
    end;
    endif
    $LATEST(r, *)$ := 0;
    $ENGAGER(r)$ := 0;
```

```
        $TIMES(r) := 0$;
        $PRIOR(r) := 0$;
        $LASTREP(r) := 0$;
    end;
    endif
    for $F := 1$ to $N$ do
    begin
                                    (* for all *queries* remaining, send an *inform* *)
                                    (* to the *query* initiator, *)
                                    (* and cancel any outstanding *queries* *)
        if $ENGAGER(F) \neq 0$
        then begin
            $TIMES(F) := T$;
            $LASTREP(F) := 0$;
            if $r \neq F$
            then send *inform* $I(P_r, P_F, LATEST(F,2), T)$;
            endif
                                    (* chop off the query trace list *)
                                    (* so $P_r$ is the last process *)
            $LATEST(F, *) :=$ **CHP** $(LATEST(F, *))$;
            if $TARGET(F) \neq 0$
            then begin
                if $EDGETYPE(F)$
                then begin
                    send *cancel* $C(P_r, TARGET(F),$ **SIG** $(LATEST(F, *)))$;
                    $EDGETYPE(F) :=$ FALSE;
                end;
                endif
                $TARGET(F) := 0$;
            end;
            endif
        end;
        endif
    end;
```

(2) When a process P_r receives a *query* $Q(P_k, P_r, U, T, P_i, M_i, ..., P_j, M_j)$, the following occurs:

```
                                    (* ensure the *query* is valid *)
    $FLAG := 0$;
    if $ENGAGER(i) = 0$
    then $FLAG := 2$;
    endif
    $v :=$ **LNG** $(QTL)$;
    $vv :=$ **LNG** $($**SIG** $(LATEST(i, *)))$;
    $l := 0$;
    while $(FLAG = 0)$ AND $(l < v)$ AND $(l < vv)$ do
    begin
```

```
          $l := l+2$;
          if $QTL(l) > LATEST(i, l)$
          then $FLAG := 1$
          else if $QTL(l) < LATEST(i, l)$
               then $FLAG := -1$;
               endif
          endif
     end;
     if $FLAG = 1$
     then begin
                                        (* send a cancel first to stop the old query *)
          if $TARGET(i) \neq 0$
          then begin
               if $EDGETYPE(i)$
               then begin
                    send cancel C($P_r$, $TARGET(i)$, CHP ($LATEST(i, *)$ ));
                    $EDGETYPE$ := FALSE;
               end;
               endif
               $TARGET(i) := 0$;
          end;
          endif
                                        (* wipe out the old query *)
          $LATEST(i, *) := 0$;
          $LASTREP(i) := 0$;
          $ENGAGER(i) := 0$;
          $PRIOR(i) := 0$;
          $TIMES(i) := 0$;
     end;
     endif
     if $FLAG \geq 0$
     then begin
          if $STATE$
          then begin
                                        (* if $P_r$ is executing *)
               if $P_r$ is already in the $QTL$
               then discard the query
               else begin
                                        (* store a copy of the query *)
                                        (* with $P_r$ added to the query trace list *)
                    $LATEST(i, *) := QTL$;
                    $LATEST(i, v+1) := P_r$;
                    $LATEST(i, v+2) := 1$;
                    $T$ := clocktime;
                    $ENGAGER(i) := P_k$;
                    $TIMES(i) := T$;
                    $PRIOR(i) := U$;
                                        (* since $P_r$ is executing, send an inform *)
```

```
                                  (* to the query computation initiator *)
                if $P_r \neq P_i$
                then send *inform* I($P_r$, $P_i$, $LATEST(i,2)$, $T$);
                endif
            end;
            endif
        end
        else begin
            if $DEADLK$
            then begin
                                  (* if $P_r$ is deadlocked, combine the reachable set *)
                                  (* with the query trace list and send a *reply* *)
                if $P_r$ is not in the $QTL$
                then **COMBIN**($QTL$, $P_r$);
                endif
                send *reply* R($P_r$, $P_k$, $T$, $QTL$);
            end
            else begin
                                  (* if $P_r$ is idle, find the first process in the *)
                                  (* dependent set which is not in the $QTL$ *)
                if $P_r$ is in the $QTL$
                then begin
                    if ($WAIT(i)$) OR (**SEQCHK**($QTL$, $P_r$) = 0)
                    then send *reply* R($P_r$, $P_k$, $T$, $QTL$);
                    endif
                end
                else begin
                    $WAIT(i)$ := TRUE;
                    $QTL(v+1) := P_r$;
                    $QTL(v+2) := 1$;
                    **NXTQRY**($P_r$, 0, $P_k$, $P_S$);
                    if $P_S = 0$
                                  (* if every process in the dependent set *)
                                  (* does not need to be queried, then *)
                                  (* send a *reply* to $P_k$ *)
                    then send *reply* R($P_r$, $P_k$, $T$, $QTL$)
                    else begin
                                  (* otherwise send a *query* to the *)
                                  (* first process in the dependent *)
                                  (* set which needs to be queried *)
                        $ENGAGER(i) := P_k$;
                        $TIMES(i) := T$;
                        $PRIOR(i) := U$;
                        $STARVE(i) := 0$;
                        $LATEST(i, *) := QTL$;
                        $IDLTIME := T - LASTTIME$;
                        if $IDLTIME \geq T_2$
```

```
                              then begin
                                   $W := 0$;
                                   for $F := 1$ to $N$ do
                                        if $ENGAGER(F) \neq 0$ AND $PRIOR(F) > W$
                                        then $W := PRIOR(F)$;
                                        endif
                                   if **URGENT**$(P_i) \geq W$
                                   then begin
                                        send *query* $Q(P_r, P_S, U, T, QTL)$;
                                        $TARGET(i) := P_S$;
                                        if $P_S$ is in the $QTL$
                                        then $EDGETYPE(i) :=$ FALSE
                                        else $EDGETYPE(i) :=$ TRUE;
                                        endif
                                   end;
                                   endif
                              end;
                              endif
                         end;
                         endif
                    end;
                    endif
               end;
               endif
          end;
          endif
     end;
     endif
```

(3) When a process P_r receives a *reply* $R(P_k, P_r, T, P_i, M_i, \ldots, P_j, M_j)$, the following occurs:

```
     if $STATE$ OR $(TARGET(i) = 0)$
                                        (* ensure the *reply* is valid *)
     then discard the *reply*
     else begin
          $F := 1$;
          $FLAG := 0$;
          while $(FLAG = 0$ AND $LATEST(i, F) \neq 0$ AND $F < 2*N)$ do
          begin
               if $QTL(F) \neq LATEST(i, F)$
               then $FLAG := 1$;
               endif
               $F := F+1$;
          end;
          if $FLAG = 1$
          then discard the *reply*
          else begin
```

```
                        (* at this point, the reply is valid *)
$TARGET(i) := 0$;
$TIMES(i) := T$;
$LASTREP(i) := P_k$;
$LATEST(i, *) := QTL$;
**NXTQRY**$(P_r, P_k, ENGAGER(i), P_S)$;
                        (* check to see if any more *)
                        (* *queries* need to be sent *)
if $P_S = 0$
                        (* if every process has been queried *)
then begin
    if $P_r = P_i$
                        (* if $P_r$ is the query initiator, declare deadlock *)
    then begin
        DECLARE DEADLOCK for all processes in $QTL$
        $ENGAGER(r) := 0$;
        $PRIOR(r) := 0$;
        $LASTREP(r) := 0$;
        for $F := 1$ to $N$ do
        begin
            if $(ENGAGER(F) \neq 0)$ AND $(r \neq F)$
            then begin
                $QTL := LATEST(F, *)$;
                $LATEST(F, *) := 0$;
                **CHP** $(QTL)$;
                    (* shorten the $QTL$ so $P_r$ is the last process *)
                if $EDGETYPE(F)$ AND $(TARGET(F) \neq 0)$
                then begin
                    send *cancel* C$(P_r, TARGET(F), QTL)$;
                    $EDGETYPE(F) :=$ FALSE;
                end;
                endif
                **COMBIN** $(QTL, P_r)$;
                send *reply* R$(P_r, ENGAGER(F), TIMES(F), QTL)$;
                $ENGAGER(F) := 0$;
                $TARGET(F) := 0$;
                $TIMES(F) := 0$;
                $PRIOR(F) := 0$;
                $LASTREP(F) := 0$;
            end;
            endif
        end;
    end
    else begin
                        (* otherwise send a *reply* to the process which *)
                        (* initially sent the *query* to $P_r$ *)
        $W := 0$;
        for $F := 1$ to $N$ do
```

```
            if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
            then $W := PRIOR(F)$;
            endif
        if $PRIOR(F) \geq W$
        then begin
            for $l := 1$ to $N$ do
            begin
                if ($ENGAGER(l) \neq 0$) AND ($TARGET(l) = 0$)
                then $STARVE(l) := STARVE(l) + 1$;
                endif
            end;
        end;
        endif
        send reply R($P_r$, $ENGAGER(i)$, $T$, $QTL$);
        $ENGAGER(i) := 0$;
        $TARGET(i) := 0$;
        $TIMES(i) := 0$;
        $PRIOR(i) := 0$;
        $LASTREP(i) := 0$;
        $LATEST(i, *) := 0$;
    end;
    endif
end;
endif
                        (* process the highest urgency queries *)
$IDLTIME := T - LASTTIME$;
if $IDLTIME \geq T_2$
then begin
    $W := 0$;
    for $F := 1$ to $N$ do
    begin
        if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
        then $W := PRIOR(F)$;
        endif
    end;
    for $F := 1$ to $N$ do
    begin
        if ($ENGAGER(F) \neq 0$)
        then begin
            if (URGENT ($F$) $\geq W$) AND ($TARGET(F) = 0$)
            then begin
                NXTQRY($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                $QTL := LATEST(F, *)$;
                $T := TIMES(F)$;
                send query Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
                $TARGET(i) := P_S$;
                if $P_S$ is already in the $QTL$
                then $EDGETYPE(i) :=$ FALSE
```

```
                    else $EDGETYPE(i)$ := TRUE;
                    endif
                end;
                endif
            end;
            endif
        end;
    end;
    endif
  end;
  endif
end;
endif
```

(4) When a process P_r receives a *cancel* C(P_k, P_r, P_i, M_i, ..., P_j, M_j), the following occurs:

```
if $ENGAGER(i) = 0$
                                    (* ensure the cancel is valid *)
then discard the cancel
else begin
    $v$ := LNG ($QTL$);
    $F$ := 1;
    $FLAG$ := 0;
    while ($FLAG = 0$ AND $F \leq v$) do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG$ := 1;
        endif
        $F := F+1$;
    end;
    if $FLAG = 1$
    then discard the cancel
    else begin
                                    (* if the cancel is valid and $P_r$ has an *)
                                    (* outstanding query for the same query *)
                                    (* initiator, send the cancel to the *)
                                    (* target process and check if the starvation *)
                                    (* count needs to be increased *)
        if $TARGET(i) \neq 0$
        then begin
            $W$ := 0;
            for $F$ := 1 to $N$ do
                if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
                then $W$ := $PRIOR(F)$;
                endif
            if $PRIOR(i) \geq W$
            then begin
```

```
                for $l$ := 1 to $N$ do
                begin
                    if ($ENGAGER(l) \neq 0$) AND ($TARGET(l) = 0$)
                    then $STARVE(l) := STARVE(l) + 1$;
                    endif
                end;
            end;
            endif
            if $EDGETYPE(i)$
            then begin
                send *cancel* C($P_{r,}$ $TARGET(i)$, $QTL$);
                $EDGETYPE(i)$ := FALSE;
            end;
            endif
            $TARGET(i) := 0$;
        end;
        endif
        $ENGAGER(i) := 0$;
        $TIMES(i) := 0$;
        $PRIOR(i) := 0$;
        $LASTREP(i) := 0$;
        $LATEST(i, *) := 0$;
                                (* if idle, then do highest priority queries *)
        if NOT $STATE$
        then begin
            $IDLTIME := SYSTIME - LASTTIME$;
            if $IDLTIME \geq T_2$
            then begin
                $W := 0$;
                for $F$ := 1 to $N$ do
                    if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
                    then $W := PRIOR(F)$;
                    endif
                for $F$ := 1 to $N$ do
                begin
                    if $ENGAGER(F) \neq 0$
                    then begin
                        if (**URGENT** $(F) \geq W$) AND ($TARGET(F) = 0$)
                        then begin
                            **NXTQRY**($P_{r,}$ $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                            $QTL := LATEST(F, *)$;
                            $T := TIMES(F)$;
                            send *query* Q($P_{r,}$ $P_{S,}$ $PRIOR(F)$, $T$, $QTL$);
                            $TARGET(F) := P_S$;
                            if $P_S$ is already in the $QTL$
                            then $EDGETYPE(F)$ := FALSE
                            else $EDGETYPE(F)$ := TRUE;
                            endif
```

```
                              end;
                              endif
                          end;
                          endif
                      end;
                  end;
                  endif
              end;
              endif
          end;
          endif
      end;
      endif
```

(5) When a process P_r receives an *inform* I(P_k, P_r, M, T), the following occurs:

```
if ($M = LATEST(r, 2)$)
then if $T > LASTTIME$
    then $LASTTIME := T$;
    endif
endif
```

(6) When a process P_k sends a message to process P_r, no query computation action is required.

(7) When an executing process P_k changes state to idle, the following occurs:

```
                                  (* at this point, the array DEPENDENT *)
                                  (* shows which processes are in the *)
                                  (* dependent set of $P_k$ *)
                                  (* $P_k$ changes state to idle *)
$STATE$ := FALSE;
$LASTTIME$ := clocktime;
$IDLE$ := clocktime;
                                  (* take care of all the queries being held *)
                                  (* process them as if they had just been received *)
for $G$ := 1 to $N$ do
begin
    if $ENGAGER(G) \neq 0$
    then begin
        $v$ := LNG ( SIG ($LATEST(G, *)$));
        $LATEST(G, v) := LATEST(G, v)+1$;
        $T := TIMES(G)$;
        $WAIT(G)$ := TRUE;
        NXTQRY($P_k$, 0, $ENGAGER(G)$, $P_S$);
        if $P_S = 0$
        then begin
            send reply R($P_k$, $ENGAGER(G)$, $T$, SIG ($LATEST(G, *)$));
            $ENGAGER(G) := 0$;
```

```
                $LATEST(G, *) := 0$;
                $TARGET(G) := 0$;
                $TIMES(G) := 0$;
                $PRIOR(G) := 0$;
                $LASTREP(G) := 0$;
            end;
            endif
        end;
        endif
    end;
```

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, the following occurs:

```
                                        (* initiate a new query computation *)
    $WAIT(i) :=$ TRUE;
    $PRIOR(i) :=$ priority to be assigned the query computation
    $ENGAGER(i) := i$;
    $NEWQRYNUM := NEWQRYNUM + 1$;
    $LATEST(i, 1) := i$;
    $LATEST(i, 2) := NEWQRYNUM$;
    $TIMES(i) := LASTTIME$;
                                        (* find the highest priority query *)
    $W := 0$;
    for $F := 1$ to $N$ do
    begin
        if $(ENGAGER(F) \neq 0)$ AND $(PRIOR(F) > W)$
        then $W := PRIOR(F)$;
        endif
    end;
                                        (* if the priority of $P_i$ is as high as the *)
                                        (* highest priority query, then send the query *)
    if URGENT $(i) \geq W)$
    then begin
        $QTL := LATEST(i, *)$;
        NXTQRY$(P_i, 0, 0, P_S$;
        if $P_S = 0$
                                        (* by assumption, processes are not terminated *)
                                        (* actually, this is a one-process deadlock *)
        then TERMINATE THE PROCESS
        else begin
            send query Q$(P_i, P_S, PRIOR(i), LASTTIME, QTL)$;
            $TARGET(i) := P_S$;
            $EDGETYPE(i) :=$ TRUE;
        end;
        endif
    end;
    endif
```

(9) When a process P_i becomes idle for time T_2 (where $T_2 < T_1$) since changing state from executing to idle, the following occurs:

```
W := 0;
for F := 1 to N do
    if (ENGAGER(F) ≠ 0) AND (PRIOR(F) > W)
    then W := PRIOR(F);
    endif
for F := 1 to N do
begin
    if ENGAGER(F) ≠ 0
    then begin
        if (URGENT (F) ≥ W) AND (TARGET(F) = 0)
        then begin
            NXTQRY(P_r, LASTREP(F), ENGAGER(F), P_S);
            QTL := LATEST(F, *);
            T := TIMES(F);
            send query Q(P_r, P_S, PRIOR(F), T, QTL);
            TARGET(F) := P_S;
            if P_S is already in the QTL
            then EDGETYPE(F) := FALSE
            else EDGETYPE(F) := TRUE;
            endif
        end;
        endif
    end;
    endif
end;
```

Appendix D.

Code for Algorithm IV

In the code shown below, QTL represents the query trace list of the current query computation. It is assumed that the QTL is variable in length and is always as short as possible. The following subprograms are used in the code shown below for Algorithm IV.

COMBIN Procedure **COMBIN** accepts a query trace list (QTL) and adds to it any process and its sequence number which is in the reachable set of the deadlocked calling process.

NXTQRY Procedure **NXTQRY** determines the next process in the dependent set which must be queried for a given query computation. It returns in its fourth argument P_S a value of zero if no more processes in the dependent set need to be queried, or else the process number of the next process to receive the query.

SEQCHK Procedure **SEQCHK** accepts a query trace list and a process id. It returns the sequence number associated with the process id. The value zero is returned if the process is not in the query trace list.

URGENT Function **URGENT** determines the urgency of a query based on its priority and the number of replies and cancels received by the process holding the query since the query was received.

UPDATR Function **UPDATR** updates the arrays *VERIFIED* and *RECENT* whenever a *query* or *reply* is received. The inputs to **UPDATR** are a query trace list, the calling process (which just received the *query* or *reply*) and the identity of the process which sent the *query* or *reply*.

CHKRPY	Procedure **CHKRPY** is called by a process to check to see if a *reply* can be sent for any held query computations. It is possible that information learned from another query computation can change the status of a held query computation so that the processes in the dependent set remaining to be queried no longer need to be queried, and hence a *reply* can be sent. **CHKRPY** takes care of sending the *reply*.
SIG	Procedure **SIG** accepts a one-dimensional array which holds a query trace list. **SIG** returns as a query trace list the portion of the array which is non-zero (ie the significant portion).
CHP	Procedure **CHP** accepts a query trace list and returns a query trace list which has been shortened so that the calling process is the last process in the query trace list.
LNG	Procedure **LNG** accepts a query trace list and returns its length. Each process id/sequence number pair counts as length 2.

Following is the pseudo-code for Algorithm IV.

(1) When an idle process P_r receives a message from process P_k and begins executing, the following occurs:

```
T := clocktime;
                                        (* process changes state to executing *)
STATE := TRUE;
for F := 1 to N do
    STARVE(F) := 0;
RECENT(r) := RECENT(r) + 1;
VERIFIED(r) := RECENT(r);
if ENGAGER(r) ≠ 0
then begin
    if TARGET(r) ≠ 0
```

```
                                (* delete the query initiated by $P_r$ *)
        then begin
            if $EDGETYPE(r)$
            then begin
                send cancel C($P_r$, $TARGET(r)$, SIG ($LATEST(r, *)$));
                                (* trailing zeros were truncated *)
                $EDGETYPE(r)$ := FALSE;
            end;
            endif
            $TARGET(r)$ := 0;
        end;
        endif
        $LATEST(r, *)$ := 0;
        $ENGAGER(r)$ := 0;
        $TIMES(r)$ := 0;
        $PRIOR(r)$ := 0;
        $LASTREP(r)$ := 0;
    end;
    endif
    for $F$ := 1 to $N$ do
    begin
                                (* for all queries remaining, send an inform *)
                                (* to the query initiator, *)
                                (* and cancel any outstanding queries *)
        if $ENGAGER(F) \neq 0$
        then begin
            $TIMES(F)$ := $T$;
            $LASTREP(F)$ := 0;
            if $r \neq F$
            then send inform I($P_r$, $P_F$, $LATEST(F,2)$, $T$);
            endif
                                (* chop off the query trace list *)
                                (* so $P_r$ is the last process *)
            $LATEST(F, *)$ := CHP ($LATEST(F, *)$);
            if $TARGET(F) \neq 0$
            then begin
                if $EDGETYPE(F)$
                then begin
                    send cancel C($P_r$, $TARGET(F)$, SIG ($LATEST(F, *)$));
                    $EDGETYPE(F)$ := FALSE;
                end;
                endif
                $TARGET(F)$ := 0;
            end;
            endif
        end;
        endif
    end;
```

(2) When a process P_r receives a *query* Q(P_k, P_r, U, T, P_i, M_i, ..., P_j, M_j), the following occurs:

```
                                        (* ensure the query is valid *)
FLAG := 0;
if ENGAGER(i) = 0
then FLAG := 2;
endif
v := LNG (QTL);
vv := LNG (SIG (LATEST(i, *)));
l := 0;
while (FLAG = 0) AND (l < v) AND (l < vv) do
begin
    l := l+2;
    if QTL(l) > LATEST(i, l)
    then FLAG := 1
    else if QTL(l) < LATEST(i, l)
        then FLAG := -1;
        endif
    endif
end;
if FLAG = 1
then begin
                                        (* send a cancel first to stop the old query *)
    if TARGET(i) ≠ 0
    then begin
        if EDGETYPE(i)
        then begin
            send cancel C(P_r, TARGET(i), CHP (LATEST(i, *) ));
            EDGETYPE := FALSE;
        end;
        endif
        TARGET(i) := 0;
    end;
    endif
                                        (* wipe out the old query *)
    LATEST(i, *) := 0;
    LASTREP(i) := 0;
    ENGAGER(i) := 0;
    PRIOR(i) := 0;
    TIMES(i) := 0;
end;
endif
if FLAG ≥ 0
then begin
    UPDATR(QTL, r, k);
    if STATE
    then begin
```

```
                        (* if $P_r$ is executing *)
        if $P_r$ is already in the $QTL$
        then discard the *query*
        else begin
                        (* store a copy of the *query* *)
                        (* with $P_r$ added to the query trace list *)
            $LATEST(i, *) := QTL$;
            $LATEST(i, v+1) := P_r$;
            $LATEST(i, v+2) := RECENT(r)$;
            $T$ := clocktime;
            $ENGAGER(i) := P_k$;
            $TIMES(i) := T$;
            $PRIOR(i) := U$;
                        (* since $P_r$ is executing, send an *inform* *)
                        (* to the query computation initiator *)
            if $P_r \neq P_i$
            then send *inform* I($P_r, P_i, LATEST(i,2), T$);
            endif
        end;
        endif
    end
    else begin
        **CHKRPY**;
        if $DEADLK$
        then begin
                        (* if $P_r$ is deadlocked, combine the reachable set *)
                        (* with the query trace list and send a *reply* *)
            if $P_r$ is not in the $QTL$
            then **COMBIN**($QTL, P_r$);
            endif
            send *reply* R($P_r, P_k, T, QTL$);
        end
        else begin
                        (* if $P_r$ is idle, find the first process in the *)
                        (* dependent set which is not in the $QTL$ *)
            if $P_r$ is in the $QTL$
            then begin
                if (**SEQCHK**($QTL, P_r$) = $RECENT(r)$)
                then send *reply* R($P_r, P_k, T, QTL$);
                endif
            end
            else begin
                $QTL(v+1) := P_r$;
                $QTL(v+2) := RECENT(r)$;
                **NXTQRY**($P_r$, 0, $P_k, P_S$);
                if $P_S = 0$
                        (* if every process in the dependent set *)
                        (* does not need to be queried, then *)
                        (* send a *reply* to $P_k$ *)
            then send *reply* R($P_r$, $P_k$, $T$, $QTL$)
            else begin
                        (* otherwise send a *query* to the *)
                        (* first process in the dependent *)
                        (* set which needs to be queried *)
                $ENGAGER(i) := P_k$;
                $TIMES(i) := T$;
                $PRIOR(i) := U$;
                $STARVE(i) := 0$;
                $LATEST(i, *) := QTL$;
            end;
            endif
        end;
        endif
        $IDLTIME := T - LASTTIME$;
        if $IDLTIME > T_2$
        then begin
            $W := 0$;
            for $F := 1$ to $N$ do
                if $ENGAGER(F) \neq 0$ AND $PRIOR(F) > W$
                then $W := PRIOR(F)$;
                endif
            for $F := 1$ to $N$ do
                if $ENGAGER(F) \neq 0$
                then begin
                    if **URGENT** ( $P_F$ ) $\geq W$ AND $TARGET(F) = 0$
                    then begin
                        **NXTQRY**($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                        send *query* Q($P_r$, $P_S$, $U$, $T$, $QTL$);
                        $TARGET(i) := P_S$;
                        if $P_S$ is in the $QTL$
                        then $EDGETYPE(i) :=$ FALSE
                        else $EDGETYPE(i) :=$ TRUE;
                        endif
                    end;
                    endif
                end;
                endif
            end;
            endif
        end;
        endif
    end;
    endif
end;
endif
```

```
end;
endif
```

(3) When a process P_r receives a *reply* R($P_k, P_r, T, P_i, M_i, ..., P_j, M_j$), the following occurs:

```
if STATE OR (TARGET(i) = 0)
                                (* ensure the reply is valid *)
then discard the reply
else begin
    F := 1;
    FLAG := 0;
    while (FLAG = 0 AND LATEST(i, F) ≠ 0 AND F < 2*N) do
    begin
        if QTL(F) ≠ LATEST(i, F)
        then FLAG := 1;
        endif
        F := F +1;
    end;
    if FLAG = 1
    then discard the reply
    else begin
                                (* at this point, the reply is valid *)
        UPDATR(QTL, P_r, P_k);
        TARGET(i) := 0;
        TIMES(i) := T;
        LASTREP(i) := P_k;
        LATEST(i, *) := QTL;
        NXTQRY(P_r, P_k, ENGAGER(i), P_S);
                                (* check to see if any more *)
                                (* queries need to be sent *)
        if P_S = 0
                                (* if every process has been queried *)
        then begin
            if P_r = P_i
                                (* if P_r is the query initiator, declare deadlock *)
            then begin
                DECLARE DEADLOCK for all processes in QTL
                ENGAGER(r) := 0;
                PRIOR(r) := 0;
                LASTREP(r) := 0;
                for F := 1 to N do
                begin
                    if (ENGAGER(F) ≠ 0) AND (r ≠ F) AND
                        ((TARGET(F) = 0) OR (NOT EDGETYPE(F)))
                    then begin
                        QTL := LATEST(F, *);
                        LATEST(F, *) := 0;
                **COMBIN**$(QTL, P_r)$;
                send *reply* R$(P_r, ENGAGER(F), TIMES(F), QTL)$;
                $ENGAGER(F) := 0$;
                $TARGET(F) := 0$;
                $TIMES(F) := 0$;
                $PRIOR(F) := 0$;
                $LASTREP(F) := 0$;
            end;
            endif
        end;
    end
    else begin
                    (* otherwise send a *reply* to the process which *)
                    (* initially sent the *query* to $P_r$ *)
        $W := 0$;
        for $F := 1$ to $N$ do
            if $(ENGAGER(F) \neq 0)$ AND $(PRIOR(F) > W)$
            then $W := PRIOR(F)$;
            endif
        if $PRIOR(F) \geq W$
        then begin
            for $l := 1$ to $N$ do
            begin
                if $(ENGAGER(l) \neq 0)$ AND $(TARGET(l) = 0)$
                then $STARVE(l) := STARVE(l) + 1$;
                endif
            end;
        end;
        endif
        send *reply* R$(P_r, ENGAGER(i), T, QTL)$;
        $ENGAGER(i) := 0$;
        $TARGET(i) := 0$;
        $TIMES(i) := 0$;
        $PRIOR(i) := 0$;
        $LASTREP(i) := 0$;
        $LATEST(i, *) := 0$;
    end;
    endif
end;
endif
                    (* check for held queries to reply to *)
**CHKRPY**;
                    (* process the highest urgency queries *)
$IDLTIME := T - LASTTIME$;
if $IDLTIME \geq T_2$
then begin
    $W := 0$;
    for $F := 1$ to $N$ do
```

```
            begin
                if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
                then $W := PRIOR(F)$;
                endif
            end;
            for $F := 1$ to $N$ do
            begin
                if ($ENGAGER(F) \neq 0$)
                then begin
                    if (**URGENT** $(F) \geq W$) AND ($TARGET(F) = 0$)
                    then begin
                        **NXTQRY**($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                        $QTL := LATEST(F, *)$;
                        $T := TIMES(F)$;
                        send *query* Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
                        $TARGET(i) := P_S$;
                        if $P_S$ is already in the $QTL$
                        then $EDGETYPE(i) :=$ FALSE
                        else $EDGETYPE(i) :=$ TRUE;
                        endif
                    end;
                    endif
                end;
                endif
            end;
        end;
        endif
    end;
    endif
end;
endif
```

(4) When a process P_r receives a *cancel* $C(P_k, P_r, P_i, M_i, ..., P_j, M_j)$, the following occurs:

```
if $ENGAGER(i) = 0$
                                    (* ensure the cancel is valid *)
then discard the cancel
else begin
    $v :=$ **LNG** ($QTL$);
    $F := 1$;
    $FLAG := 0$;
    while ($FLAG = 0$ AND $F < v$) do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG := 1$;
        endif
        $F := F+1$;
```

```
end;
if $FLAG = 1$
then discard the *cancel*
else begin
                              (* if the *cancel* is valid and $P_r$ has an *)
                              (* *outstanding* *query* for the same query *)
                              (* initiator, send the *cancel* to the *)
                              (* target process and check if the starvation *)
                              (* count needs to be increased *)
    if $TARGET(i) \neq 0$
    then begin
        $W := 0$;
        for $F := 1$ to $N$ do
            if $(ENGAGER(F) \neq 0)$ AND $(PRIOR(F) > W)$
            then $W := PRIOR(F)$;
            endif
        if $PRIOR(i) \geq W$
        then begin
            for $l := 1$ to $N$ do
            begin
                if $(ENGAGER(l) \neq 0)$ AND $(TARGET(l) = 0)$
                then $STARVE(l) := STARVE(l) + 1$;
                endif
            end;
        end;
        endif
        if $EDGETYPE(i)$
        then begin
            send *cancel* C($P_r$, $TARGET(i)$, $QTL$);
            $EDGETYPE(i)$ := FALSE;
        end;
        endif
        $TARGET(i) := 0$;
    end;
    endif
    $ENGAGER(i) := 0$;
    $TIMES(i) := 0$;
    $PRIOR(i) := 0$;
    $LASTREP(i) := 0$;
    $LATEST(i, *) := 0$;
                              (* if idle, then do highest priority queries *)
    if NOT $STATE$
    then begin
        **CHKRPY**;
        $IDLTIME := SYSTIME - LASTTIME$;
        if $IDLTIME > T_2$
        then begin
            $W := 0$;
```

```
                    for $F$ := 1 to $N$ do
                        if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
                        then $W := PRIOR(F)$;
                        endif
                    for $F$ := 1 to $N$ do
                    begin
                        if $ENGAGER(F) \neq 0$
                        then begin
                            if (**URGENT** $(F) \geq W$) AND ($TARGET(F) = 0$)
                            then begin
                                **NXTQRY**($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                                $QTL := LATEST(F, *)$;
                                $T := TIMES(F)$;
                                send *query* Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
                                $TARGET(F) := P_S$;
                                if $P_S$ is already in the $QTL$
                                then $EDGETYPE(F)$ := FALSE
                                else $EDGETYPE(F)$ := TRUE;
                                endif
                            end;
                            endif
                        end;
                        endif
                    end;
                end;
                endif
            end;
            endif
        end;
        endif
    end;
    endif
```

(5) When a process P_r receives an *inform* I(P_k, P_r, M, T), the following occurs:

```
    if ($M = LATEST(r, 2)$)
    then if $T > LASTTIME$
        then $LASTTIME := T$;
        endif
    endif
```

(6) When a process P_k sends a message to process P_r, no query computation action is required.

(7) When an executing process P_k changes state to idle, the following occurs:

```
                                (* at this point, the array $DEPENDENT$ *)
                                (* shows which processes are in the *)
                                (* dependent set of $P_k$ *)
```

```
                                        (* $P_k$ changes state to idle *)
$STATE$ := FALSE;
$LASTTIME$ := clocktime;
$IDLE$ := clocktime;
                                        (* take care of all the *queries* being held *)
                                        (* process them as if they had just been received *)
for $G$ := 1 to $N$ do
begin
    if $ENGAGER(G) \neq 0$
    then begin
        $v$ := **LNG** ( **SIG** ($LATEST(G, *)$));
        $LATEST(G, v) := RECENT(k)$;
        $T := TIMES(G)$;
        **NXTQRY**($P_k$, 0, $ENGAGER(G)$, $P_S$);
        if $P_S = 0$
        then begin
            send *reply* R($P_k$, $ENGAGER(G)$, $T$, **SIG** ($LATEST(G, *)$));
            $ENGAGER(G) := 0$;
            $LATEST(G, *) := 0$;
            $TARGET(G) := 0$;
            $TIMES(G) := 0$;
            $PRIOR(G) := 0$;
            $LASTREP(G) := 0$;
        end;
        endif
    end;
    endif
end;
```

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, the following occurs:

```
                                        (* initiate a new query computation *)
$PRIOR(i)$ := priority to be assigned the query computation
$ENGAGER(i) := i$;
$LATEST(i, 1) := i$;
$LATEST(i, 2) := RECENT(i)$
$TIMES(i) := LASTTIME$;
                                        (* find the highest priority *query* *)
$W := 0$;
for $F$ := 1 to $N$ do
begin
    if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
    then $W := PRIOR(F)$;
    endif
end;
                                        (* if the priority of $P_i$ is as high as the *)
                                        (* highest priority *query*, then send the *query* *)
```

```
if URGENT $(i) \geq W)$
then begin
    $QTL := LATEST(i, *)$;
    NXTQRY$(P_i, 0, 0, P_S$;
    if $P_S = 0$
                                    (* by assumption, processes are not terminated *)
                                    (* actually, this is a one-process deadlock *)
    then TERMINATE THE PROCESS
    else begin
        send query Q($P_i, P_S, PRIOR(i), LASTTIME, QTL$);
        $TARGET(i) := P_S$;
        $EDGETYPE(i)$ := TRUE;
    end;
    endif
end;
endif
```

(9) When a process P_i becomes idle for time T_2 (where $T_2 < T_1$) since changing state from executing to idle, the following occurs:

```
CHKRPY;
$W := 0$;
for $F := 1$ to $N$ do
    if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
    then $W := PRIOR(F)$;
    endif
for $F := 1$ to $N$ do
begin
    if $ENGAGER(F) \neq 0$
    then begin
        if (URGENT $(F) \geq W$) AND ($TARGET(F) = 0$)
        then begin
            NXTQRY$(P_r, LASTREP(F), ENGAGER(F), P_S)$;
            $QTL := LATEST(F, *)$;
            $T := TIMES(F)$;
            send query Q($P_r, P_S, PRIOR(F), T, QTL$);
            $TARGET(F) := P_S$;
            if $P_S$ is already in the $QTL$
            then $EDGETYPE(F)$ := FALSE
            else $EDGETYPE(F)$ := TRUE;
            endif
        end;
        endif
    end;
    endif
end;
```

Appendix E.

Code for Algorithm V

In the code shown below, QTL represents the query trace list of the current query computation. It is assumed that the QTL is variable in length and is always as short as possible. The following subprograms are used in the code shown below for Algorithm V.

COMBIN Procedure **COMBIN** accepts a query trace list (QTL) and adds to it any process (together with its sequence number and boolean flag) which is in the reachable set of the deadlocked calling process.

NXTQRY Procedure **NXTQRY** determines the next process in the dependent set which must be queried for a given query computation. It returns in its fourth argument P_S a value of zero if no more processes in the dependent set need to be queried, or else the process number of the next process to receive the query.

SEQCHK Procedure **SEQCHK** accepts a query trace list and a process id. It returns the sequence number associated with the process id. The value zero is returned if the process is not in the query trace list.

URGENT Function **URGENT** determines the urgency of a query based on its priority and the number of replies and cancels received by the process holding the query since the query was received.

UPDATR Function **UPDATR** updates the arrays *VERIFIED* and *RECENT* whenever a *query* or *reply* is received. The inputs to **UPDATR** are a query trace list, the calling process (which just received the *query* or *reply*) and the identity of the process which sent the *query* or *reply*.

REPUPD Procedure **REPUPD**, when called by P_r, updates the array $LATEST(r, *)$ with information from the query trace list of a *reply* received by P_r. The flags of any process up to and including P_r are changed to reflect the way they are in the query trace list. The flag of any process which appears after P_r is not changed. Any process which appears in the query trace list but not in the array $LATEST(r, *)$ is added, together with its sequence number and flag as they appear in the query trace list.

CHKRPY Procedure **CHKRPY** is called by a process to check to see if a *reply* can be sent for any held query computations. It is possible that information learned from another query computation can change the status of a held *query computation* so that the processes in the dependent set remaining to be queried no longer need to be queried, and hence a *reply* can be sent. **CHKRPY** takes care of sending the *reply*.

FGFIXR Procedure **FGFIXR** accepts as input a query trace list, the calling process P_r, and the array $LATEST(r, *)$. It is called by a process which knows it is deadlocked and is preparing a *reply* to send to its engager. The deadlocked process first calls **COMBIN** to add the processes in its reachable set to the query trace list, and then it calls **FGFIXR** to set the flags correctly in the query trace list. **FGFIXR** finds the first process in the query trace list which is also in the reachable set of P_r. The boolean flags for all processes occurring after this process in the query trace list are set to TRUE.

FGCHKR	Procedure **FGCHKR** accepts as input a query trace list, the calling process P_r, and the array *DEPENDENT* belonging to P_r. It is called by a process which has just received an engaging *query*. The process P_r calls **FGCHKR** to set the flags correctly in the query trace list. **FGCHKR** finds the first process in the query trace list which is also in the dependet set of P_r. The boolean flags for all processes occurring after this process in the query trace list are set to TRUE.
CHKMDS	Procedure **CHKMDS** accepts as input a query trace list and the calling process P_r. It is called by P_r when it has just received its last *reply*. The process P_r calls **CHKMDS** to set the flags correctly in the query trace list and to check for membership in a minimal deadlocked set. **CHKMDS** first finds the process P_r in the query trace list. If its boolean flag is FALSE (–), then P_r is a member of a minimal deadlocked set. If so, then all boolean flags preceeding the one for process P_r are set to TRUE (+), and the ones following it are set to show the pattern (– – – – +).
SIG	Procedure **SIG** accepts a one-dimensional array which holds a query trace list. **SIG** returns as a query trace list the portion of the array which is non-zero (ie the significant portion).
CHP	Procedure **CHP** accepts a query trace list and returns a query trace list which has been shortened so that the calling process is the last process in the query trace list.
LNG	Procedure **LNG** accepts a query trace list and returns its length. Each process id/sequence number pair counts as length 2.

ABS Procedure **ABS** is just the standard function for absolute value.

Following is the pseudo-code for Algorithm V.

(1) When an idle process P_r receives a message from process P_k and begins executing, the following occurs:

```
T := clocktime;
                                         (* process changes state to executing *)
STATE := TRUE;
for F := 1 to N do
    STARVE(F) := 0;
RECENT(r) := RECENT(r) + 1;
VERIFIED(r) := RECENT(r);
if ENGAGER(r) ≠ 0
then begin
    if TARGET(r) ≠ 0
                                         (* delete the query initiated by P_r *)
    then begin
        if EDGETYPE(r)
        then begin
            send cancel C(P_r, TARGET(r), SIG (LATEST(r, *)));
                                         (* trailing zeros were truncated *)
            EDGETYPE(r) := FALSE;
        end;
        endif
        TARGET(r) := 0;
    end;
    endif
    LATEST(r, *) := 0;
    ENGAGER(r) := 0;
    TIMES(r) := 0;
    PRIOR(r) := 0;
    LASTREP(r) := 0;
end;
endif
for F := 1 to N do
begin
                                         (* for all queries remaining, send an inform *)
                                         (* to the query initiator, *)
                                         (* and cancel any outstanding queries *)
    if ENGAGER(F) ≠ 0
    then begin
        TIMES(F) := T;
        LASTREP(F) := 0;
        if r ≠ F
        then send inform I(P_r, P_F, LATEST(F,2), T);
        endif
```

```
                                        (* chop off the query trace list *)
                                        (* so $P_r$ is the last process *)
            $LATEST(F, *)$ := CHP $(LATEST(F, *))$;
            $v$ := LNG ( SIG $(LATEST(F, *)))$;
            $LATEST(F, v)$ := − ABS $(LATEST(F, v))$;
            if $TARGET(F) \neq 0$
            then begin
                if $EDGETYPE(F)$
                then begin
                    send cancel C($P_r$, $TARGET(F)$, SIG $(LATEST(F, *)))$;
                    $EDGETYPE(F)$ := FALSE;
                end;
                endif
                $TARGET(F)$ := 0;
            end;
            endif
        end;
        endif
    end;
```

(2) When a process P_r receives a *query* Q(P_k, P_r, U, T, P_i, M_i, ..., P_j, M_j), the following occurs:

```
                                        (* ensure the query is valid *)
    $FLAG$ := 0;
    if $ENGAGER(i) = 0$
    then $FLAG$ := 2;
    endif
    $v$ := LNG $(QTL)$;
    $vv$ := LNG (SIG $(LATEST(i, *)))$;
    $l$ := 0;
    while $(FLAG = 0)$ AND $(l < v)$ AND $(l < vv)$ do
    begin
        $l$ := $l$+2;
        if ABS $(QTL(l))$ > ABS $(LATEST(i, l))$
        then $FLAG$ := 1
        else if ABS $(QTL(l))$ < ABS $(LATEST(i, l))$
            then $FLAG$ := -1;
            endif
        endif
    end;
    if $FLAG = 1$
    then begin
                                        (* send a cancel first to stop the old query *)
        if $TARGET(i) \neq 0$
        then begin
            if $EDGETYPE(i)$
            then begin
```

```
                send cancel C($P_r$, $TARGET(i)$, CHP ($LATEST(i, *)$ ));
                $EDGETYPE$ := FALSE;
            end;
            endif
            $TARGET(i)$ := 0;
        end;
        endif
                                    (* wipe out the old query *)
        $LATEST(i, *)$ := 0;
        $LASTREP(i)$ := 0;
        $ENGAGER(i)$ := 0;
        $PRIOR(i)$ := 0;
        $TIMES(i)$ := 0;
    end;
    endif
    if $FLAG \geq 0$
    then begin
        UPDATR($QTL$, $r$, $k$);
        if $STATE$
        then begin
                                    (* if $P_r$ is executing *)
            if $P_r$ is already in the $QTL$
            then discard the query
            else begin
                                    (* store a copy of the query *)
                                    (* with $P_r$ added to the query trace list *)
                $LATEST(i, *)$ := $QTL$;
                $LATEST(i, v+1)$ := $P_r$;
                $LATEST(i, v+2)$ := $- RECENT(r)$;
                $T$ := clocktime;
                $ENGAGER(i)$ := $P_k$;
                $TIMES(i)$ := $T$;
                $PRIOR(i)$ := $U$;
                                    (* since $P_r$ is executing, send an inform *)
                                    (* to the query computation initiator *)
                if $P_r \neq P_i$
                then send inform I($P_r$, $P_i$, $LATEST(i,2)$, $T$);
                endif
            end;
            endif
        end
        else begin
            CHKRPY;
            if $DEADLK$
            then begin
                                    (* if $P_r$ is deadlocked, combine the reachable set *)
                                    (* with the query trace list and send a reply *)
                                    (* Also check for minimal deadlocked sets *)
```

```
                    (* and set the boolean flags correctly *)
        if $LATEST(r, 2) < 0$
        then begin
            if $TARGET(i) \neq 0$
            then begin
                **COMBIN**$(QTL, P_r)$;
                **FGFIXR**$(P_r, QTL)$;
            end
            else begin
                *inflag* := FALSE;
                for $j$ := 1 to $N$ do
                begin
                    $jj := j * 2 + 1$;
                    if $LATEST(r, jj) = P_k$
                    then *inflag* := TRUE;
                end;
                if *inflag*
                then begin
                    **COMBIN**$(QTL, P_r)$;
                    **FGFIXR**$(P_r, QTL)$;
                end
                else begin
                    the flag for $P_k$ in the $QTL$ is set to TRUE (+)
                    **COMBIN**$(QTL, P_r)$;
                end;
                endif
            end;
            endif
        end
        else begin
            the flag for $P_k$ in the $QTL$ is set to TRUE (+)
            **COMBIN**$(QTL, P_r)$;
        end;
        endif
        send *reply* R$(P_r, P_k, T, QTL)$;
    end
    else begin
                    (* if $P_r$ is idle, find the first process in the *)
                    (* dependent set which is not in the $QTL$ *)
        if $P_r$ is in the $QTL$
        then begin
            if (**SEQCHK**$(QTL, P_r) = RECENT(r)$)
            then send *reply* R$(P_r, P_k, T, QTL)$;
            endif
        end
        else begin
            $QTL(v+1) := P_r$;
```

```
          $QTL(v+2) := -RECENT(r)$;
          **FGCHKR**$(QTL, P_r)$;
          **NXTQRY**$(P_r, 0, P_k, P_S)$;
          if $P_S = 0$
                      (* if every process in the dependent set *)
                      (* does not need to be queried, then *)
                      (* send a reply to $P_k$ *)
          then begin
              **CHKMDS**$(QTL, P_r)$;
              send reply $R(P_r, P_k, T, QTL)$;
          end
          else begin
                      (* otherwise send a query to the *)
                      (* first process in the dependent *)
                      (* set which needs to be queried *)
              $ENGAGER(i) := P_k$;
              $TIMES(i) := T$;
              $PRIOR(i) := U$;
              $STARVE(i) := 0$;
              $LATEST(i, *) := QTL$;
          end;
          endif
      end;
      endif
      $IDLTIME := T - LASTTIME$;
      if $IDLTIME \geq T_2$
      then begin
          $W := 0$;
          for $F := 1$ to $N$ do
              if $ENGAGER(F) \neq 0$ AND $PRIOR(F) > W$
              then $W := PRIOR(F)$;
              endif
          for $F := 1$ to $N$ do
              if $ENGAGER(F) \neq 0$
              then begin
                  if **URGENT** $(P_F) \geq W$ AND $TARGET(F) = 0$
                  then begin
                      **NXTQRY**$(P_r, LASTREP(F), ENGAGER(F), P_S)$;
                      send query $Q(P_r, P_S, U, T, QTL)$;
                      $TARGET(i) := P_S$;
                      if $P_S$ is in the $QTL$
                      then $EDGETYPE(i) :=$ FALSE
                      else $EDGETYPE(i) :=$ TRUE;
                      endif
                  end;
                  endif
              end;
```

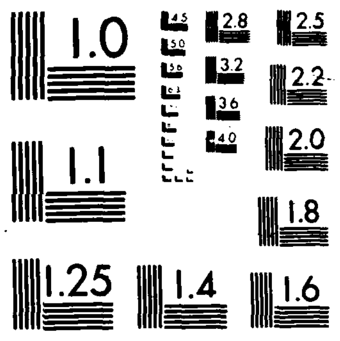
```
                    endif
                end;
                endif
            end;
            endif
        end;
        endif
    end;
    endif
end;
endif
```

(3) When a process P_r receives a *reply* R(P_k, P_r, T, P_i, M_i, ..., P_j, M_j), the following occurs:

```
if $STATE$ OR ($TARGET(i) = 0$)
                                (* ensure the reply is valid *)
then discard the reply
else begin
    $F := 1$;
    $FLAG := 0$;
    while ($FLAG = 0$ AND $LATEST(i, F) \neq 0$ AND $F < 2$ [illegible]) do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG := 1$;
        endif
        $F := F+1$;
    end;
    if $FLAG = 1$
    then discard the reply
    else begin
                                (* at this point, the reply is valid *)
        UPDATR($QTL$, $P_r$, $P_k$);
        $TARGET(i) := 0$;
        $TIMES(i) := T$;
        $LASTREP(i) := P_k$;
        REPUPD($QTL$, $P_r$);
        NXTQRY($P_r$, $P_k$, $ENGAGER(i)$, $P_S$);
                                (* check to see if any more *)
                                (* queries need to be sent *)
        if $P_S = 0$
                                (* if every process has been queried *)
        then begin
            if $P_r = P_i$
                                (* if $P_r$ is the query initiator, declare deadlock *)
                                (* deadlock can be declared when checking for *)
                                (* minimal deadlocked sets *)
            then begin
    CHKMDS ($QTL$, $P_r$);
    $ENGAGER(r) := 0$;
    $PRIOR(r) := 0$;
    $LASTREP(r) := 0$;
    for $F := 1$ to $N$ do
    begin
        if ($ENGAGER(F) \neq 0$) AND ($r \neq F$) AND
            (($TARGET(F) = 0$) OR (NOT $EDGETYPE(F)$))
        then begin
            $QTL := LATEST(F, *)$;
            $LATEST(F, *) := 0$;
            if $LATEST(r, 2) \neq 0$
            then begin
                $inflag$ := FALSE;
                for $j := 1$ to $N$ do
                begin
                    $jj := j * 2 + 1$;
                    if $LATEST(r, jj) = ENGAGER(F)$
                    then $inflag$ := TRUE;
                end;
                if $inflag$
                then begin
                    COMBIN ($QTL$, $P_r$);
                    FGFIXR ($P_r$, $QTL$);
                end
                else begin
                    the flag for the engager of $P_r$
                        in the $QTL$ is set to TRUE (+)
                    COMBIN ($QTL$, $P_r$);
                end;
                endif
            end
            else begin
                the flag for the engager of $P_r$
                    in the $QTL$ is set to TRUE (+)
                COMBIN ($QTL$, $P_r$);
            end;
            endif
            send reply R($P_r$, $ENGAGER(F)$, $TIMES(F)$, $QTL$);
            $ENGAGER(F) := 0$;
            $TARGET(F) := 0$;
            $TIMES(F) := 0$;
            $PRIOR(F) := 0$;
            $LASTREP(F) := 0$;
        end;
        endif
    end;
end
```

AD-A172 839   DISTRIBUTED DEADLOCK DETECTION FOR COMMUNICATING   3/3
PROCESSES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON
AFB OH R A ADAMS 1986 AFIT/CI/NR-86-174D

UNCLASSIFIED   F/G 9/2   NL


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
        else begin
                                (* otherwise send a reply to the process which *)
                                (* initially sent the query to $P_r$ *)
            $W := 0$;
            for $F := 1$ to $N$ do
                if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
                then $W := PRIOR(F)$;
                endif
            if $PRIOR(F) \geq W$
            then begin
                for $l := 1$ to $N$ do
                begin
                    if ($ENGAGER(l) \neq 0$) AND ($TARGET(l) = 0$)
                    then $STARVE(l) := STARVE(l) + 1$;
                    endif
                end;
            end;
            endif
            CHKMDS($QTL$, $P_r$);
            send reply R($P_r$, $ENGAGER(i)$, $T$, $QTL$);
            $ENGAGER(i) := 0$;
            $TARGET(i) := 0$;
            $TIMES(i) := 0$;
            $PRIOR(i) := 0$;
            $LASTREP(i) := 0$;
            $LATEST(i, *) := 0$;
        end;
        endif
    end;
    endif
                                (* check for held queries to reply to *)
    CHKRPY;
                                (* process the highest urgency queries *)
    $IDLTIME := T - LASTTIME$;
    if $IDLTIME \geq T_2$
    then begin
        $W := 0$;
        for $F := 1$ to $N$ do
        begin
            if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
            then $W := PRIOR(F)$;
            endif
        end;
        for $F := 1$ to $N$ do
        begin
            if ($ENGAGER(F) \neq 0$)
            then begin
                if (URGENT ($F$) $\geq W$) AND ($TARGET(F) = 0$)
```

```
                    then begin
                        NXTQRY($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                        $QTL := LATEST(F, *)$;
                        $T := TIMES(F)$;
                        send query Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
                        $TARGET(i) := P_S$;
                        if $P_S$ is already in the $QTL$
                        then $EDGETYPE(i)$ := FALSE
                        else $EDGETYPE(i)$ := TRUE;
                        endif
                    end;
                    endif
                end;
                endif
            end;
        end;
        endif
    end;
    endif
end;
endif
```

(4) When a process P_r receives a *cancel* C(P_k, P_r, P_i, M_i, ..., P_j, M_j), the following occurs:

```
if $ENGAGER(i) = 0$
                                        (* ensure the cancel is valid *)
then discard the cancel
else begin
    $v$ := LNG ($QTL$);
    $F := 1$;
    $FLAG := 0$;
    while ($FLAG = 0$ AND $F \le v$) do
    begin
        if $QTL(F) \neq LATEST(i, F)$
        then $FLAG := 1$;
        endif
        $F := F+1$;
    end;
    if $FLAG = 1$
    then discard the cancel
    else begin
                                        (* if the cancel is valid and $P_r$ has an *)
                                        (* outstanding query for the same query *)
                                        (* initiator, send the cancel to the *)
                                        (* target process and check if the starvation *)
                                        (* count needs to be increased *)
        if $TARGET(i) \neq 0$
```

```
then begin
    $W$ := 0;
    for $F$ := 1 to $N$ do
        if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
        then $W$ := $PRIOR(F)$;
        endif
    if $PRIOR(i) \geq W$
    then begin
        for $l$ := 1 to $N$ do
        begin
            if ($ENGAGER(l) \neq 0$) AND ($TARGET(l) = 0$)
            then $STARVE(l)$ := $STARVE(l) + 1$;
            endif
        end;
    end;
    endif
    if $EDGETYPE(i)$
    then begin
        send cancel C($P_r$, $TARGET(i)$, $QTL$);
        $EDGETYPE(i)$ := FALSE;
    end;
    endif
    $TARGET(i)$ := 0;
end;
endif
$ENGAGER(i)$ := 0;
$TIMES(i)$ := 0;
$PRIOR(i)$ := 0;
$LASTREP(i)$ := 0;
$LATEST(i, *)$ := 0;
                                (* if idle, then do highest priority queries *)
if NOT $STATE$
then begin
    CHKRPY;
    $IDLTIME$ := $SYSTIME - LASTTIME$;
    if $IDLTIME \geq T_2$
    then begin
        $W$ := 0;
        for $F$ := 1 to $N$ do
            if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
            then $W$ := $PRIOR(F)$;
            endif
        for $F$ := 1 to $N$ do
        begin
            if $ENGAGER(F) \neq 0$
            then begin
                if (URGENT $(F) > W$) AND ($TARGET(F) = 0$)
                then begin
```

```
                            NXTQRY($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
                            $QTL := LATEST(F, *)$;
                            $T := TIMES(F)$;
                            send query Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
                            $TARGET(F) := P_S$;
                            if $P_S$ is already in the $QTL$
                            then $EDGETYPE(F)$ := FALSE
                            else $EDGETYPE(F)$ := TRUE;
                            endif
                        end;
                        endif
                    end;
                    endif
                end;
            end;
            endif
        end;
        endif
    end;
    endif
end;
endif
```

(5) When a process P_r receives an *inform* I(P_k, P_r, M, T), the following occurs:

```
if ABS $(M)$ = ABS $(LATEST(r, 2))$
then if $T > LASTTIME$
    then $LASTTIME := T$;
    endif
endif
```

(6) When a process P_k sends a message to process P_r, no query computation action is required.

(7) When an executing process P_k changes state to idle, the following occurs:

```
                                (* at this point, the array DEPENDENT *)
                                (* shows which processes are in the *)
                                (* dependent set of $P_k$ *)
                                (* $P_k$ changes state to idle *)
$STATE$ := FALSE;
$LASTTIME$ := clocktime;
$IDLE$ := clocktime;
                                (* take care of all the queries being held *)
                                (* process them as if they had just been received *)
for $G := 1$ to $N$ do
begin
    if $ENGAGER(G) \neq 0$
    then begin
```

```
            $v$ := LNG ( SIG ($LATEST(G, *)$));
            $LATEST(G, v) := -RECENT(k)$;
            FGCHKR($LATEST(G, *), P_k$);
            $T := TIMES(G)$;
            NXTQRY($P_k$, 0, $ENGAGER(G), P_S$);
            if $P_S = 0$
            then begin
                CHKMDS($QTL, P_k$);
                send reply R($P_k$, $ENGAGER(G), T$, SIG ($LATEST(G, *)$));
                $ENGAGER(G) := 0$;
                $LATEST(G, *) := 0$;
                $TARGET(G) := 0$;
                $TIMES(G) := 0$;
                $PRIOR(G) := 0$;
                $LASTREP(G) := 0$;
            end;
            endif
        end;
        endif
    end;
```

(8) When a process P_i becomes idle for time T_1 since changing state from executing to idle, the following occurs:

```
                                    (* initiate a new query computation *)
$PRIOR(i)$ := priority to be assigned the query computation
$ENGAGER(i) := i$;
$LATEST(i, 1) := i$;
$LATEST(i, 2) := -RECENT(i)$
$TIMES(i) := LASTTIME$;
                                    (* find the highest priority query *)
$W := 0$;
for $F := 1$ to $N$ do
begin
    if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
    then $W := PRIOR(F)$;
    endif
end;
                                    (* if the priority of $P_i$ is as high as the *)
                                    (* highest priority query, then send the query *)
if URGENT ($i$) $\geq W$)
then begin
    $QTL := LATEST(i, *)$;
    NXTQRY($P_i$, 0, 0, $P_S$;
    if $P_S = 0$
                                    (* by assumption, processes are not terminated *)
                                    (* actually, this is a one-process deadlock *)
    then TERMINATE THE PROCESS
```

```
        else begin
            send query Q($P_i$, $P_S$, $PRIOR(i)$, $LASTTIME$, $QTL$);
            $TARGET(i) := P_S$;
            $EDGETYPE(i) :=$ TRUE;
        end;
        endif
    end;
    endif
```

(9) When a process P_i becomes idle for time T_2 (where $T_2 < T_1$) since changing state from executing to idle, the following occurs:

```
CHKRPY;
$W := 0$;
for $F := 1$ to $N$ do
    if ($ENGAGER(F) \neq 0$) AND ($PRIOR(F) > W$)
    then $W := PRIOR(F)$;
    endif
for $F := 1$ to $N$ do
begin
    if $ENGAGER(F) \neq 0$
    then begin
        if (URGENT $(F) \geq W$) AND ($TARGET(F) = 0$)
        then begin
            NXTQRY($P_r$, $LASTREP(F)$, $ENGAGER(F)$, $P_S$);
            $QTL := LATEST(F, *)$;
            $T := TIMES(F)$;
            send query Q($P_r$, $P_S$, $PRIOR(F)$, $T$, $QTL$);
            $TARGET(F) := P_S$;
            if $P_S$ is already in the $QTL$
            then $EDGETYPE(F) :=$ FALSE
            else $EDGETYPE(F) :=$ TRUE;
            endif
        end;
        endif
    end;
    endif
end;
```

Appendix F.

Definitions

Deadlock: A nonempty set of processes S is *deadlocked* if and only if

1) All processes in S are idle;

2) The dependent set of every process in S is a subset of S; and

3) There are no messages in transit between processes in S.

A process is *deadlocked* if and only if it belongs to some deadlocked set.

Minimal deadlocked set

A minimal deadlocked set is a deadlocked set which is either strongly connected or which is a strongly connected component [AhHo74, page 189].

Engaging query and Engager:

When a process P_j receives a query from a process P_i belonging to a query computation that P_j has not seen before (i.e. P_j is not in the QTL), then the query is called an *engaging query*, and process P_i is called the *engager*.

Engager chain: For a particular query computation, the *engager chain* consists of the query computation initiator and all processes P_k such that the following conditions hold:

1) P_k is in the dependent set of some process P_j,

2) P_j is in the engager chain,

3) P_j sent an engaging query to P_k,

4) P_j has not executed since sending the query to P_k, and

5) P_k has received, but not yet replied to the engaging query.

Each query computation has only one engager chain. If, for a particular query computation, process P is a member of the engager chain, then the engager chain of P consists of the query computation initiator and all processes in the engager chain up to but not including P.

Current query computation:

A query sent from process P to process Q belongs to the current query computation if and only if

1) P was a member of the engager chain for that query computation when it sent the query, and

2) Neither P nor any process on the engager chain of P has executed since P sent the query.

A reply sent from process Q to process P belongs to the current query computation if and only if the corresponding query sent from P to Q belongs to the query computation.

Target process: When a process P_j receives a query (not necessarily an engaging query) from a process P_i, then process P_j is the target process for that query.

Reachable set: Consider a directed graph in which nodes represent processes. A directed edge from A to B means that process A is waiting to receive a message from process B, and therefore B is in the dependent set of A. If, for some process C, there is a directed path from A to C, then C is said to be *reachable* from A, and C is in the *reachable set* of A.

Observation 1 If P is in the reachable set of Q, and Q is in the reachable set of R, then P is in the reachable set of R.

Observation 2 If P is in the reachable set of Q, then the size of the reachable set of Q is greater than or equal to the size of the reachable set of P.

Appendix G.

Examples for Deadlock Detection Algorithms

This appendix contains examples showing how the five new algorithms and the CMH algorithm detect deadlock. The first example shows the primary differences between the algorithms, although it does not show the use of the *cancel*. The second example shows in greater detail the way in which Algorithm V detects the minimal deadlocked sets.

Example 1

The first example shows how the different algorithms work when encountering an executing process and how they then go on to detect deadlock once it exists. The initial state of the system is shown in Figure 46. Process D is executing and is in the dependent sets of processes B and E. The primary events will occur in the following order:

(1) Process A will initiate a query computation.

(2) Process B will initiate a query computation.

(3) Process D will become idle and wait for processes B and E (see Figure 47).

(4) Process D will initiate a query computation.

A plausible sequence of events showing how the deadlock detection algorithms work within this scenario is shown for each algorithm.

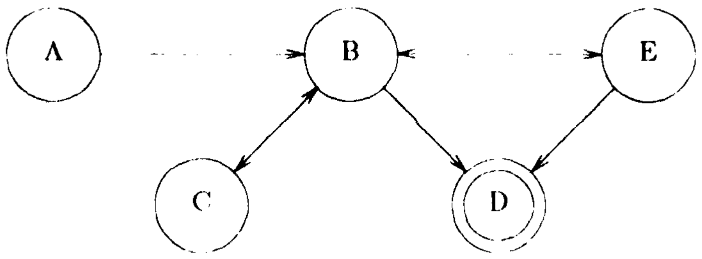


Figure 46
Initial Configuration without Deadlock

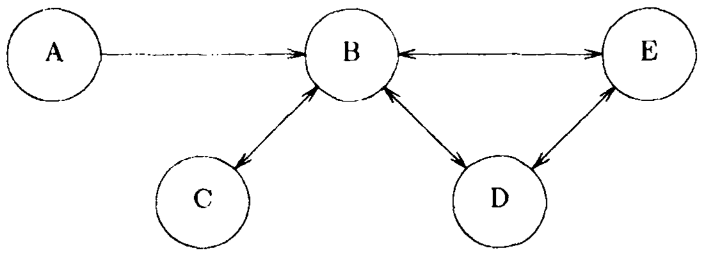


Figure 47
Final Configuration with Deadlock

Algorithm CMH

The CMH Algorithm is shown first so the new algorithms may be compared to it.

Time Action

(1) A initiates a query computation
A sends Q (A,1,A,B) to B
A sets its *NUM*(*A*) to 1

(2) B receives Q (A,1,A,B) and is idle, so
B sends Q (A,1,B,C) to C
B sends Q (A,1,B,D) to D
B sends Q (A,1,B,E) to E
B sets its *NUM*(*A*) to 3

(3) C receives Q (A,1,B,C) and is idle, so
C sends Q (A,1,C,B) to B
C sets its *NUM*(*A*) to 1

(4) D receives Q (A,1,B,D) and is executing, so
D discards the query

(5) E receives Q (A,1,B,E) and is idle, so
E sends Q (A,1,E,B) to B
E sends Q (A,1,E,D) to D
E sets its *NUM*(*A*) to 2

(6) B receives Q (A,1,C,B) and is still idle, so
B sends R (A,1,B,C) to C

(7) B receives Q (A,1,E,B) and is still idle, so
B sends R (A,1,B,E) to E

(8) D receives Q (A,1,E,D) and is executing, so
D discards the query

(9) C receives R (A,1,B,C)
C decrements its $NUM(A)$ by 1
C's $NUM(A)$ is now 0, so
C sends R (A,1,C,B) to B

(10) E receives R (A,1,B,E) and is still idle, so
E decrements its $NUM(A)$ by 1
E's $NUM(A)$ is now 1, so E waits for another reply

(11) B receives R (A,1,C,B) and is still idle, so
B decrements its $NUM(A)$ by 1
B's $NUM(A)$ is now 2, so B waits for two more replies

(12) B initiates a query computation
B sends Q (B,1,B,C) to C
B sends Q (B,1,B,D) to D
B sends Q (B,1,B,E) to E
B sets its $NUM(B)$ to 3

(13) C receives Q (B,1,B,C) and is idle, so
C sends Q (B,1,C,B) to B
C sets its $NUM(B)$ to 1

(14) D receives Q (B,1,B,D) and is executing, so
D discards the query

(15) E receives Q (B,1,B,E) and is idle, so
E sends Q (B,1,E,B) to B
E sends Q (B,1,E,D) to D
E sets its $NUM(B)$ to 2

(16) B receives Q (B,1,C,B) and is still idle, so
B sends R (B,1,B,C) to C

(17) B receives Q (B,1,E,B) and is still idle, so
B sends R (B,1,B,E) to E

(18) D receives Q (B,1,E,D) and is executing, so
D discards the query

(19) C receives R (B,1,B,C)
C decrements its $NUM(B)$ by 1
C's $NUM(B)$ is now 0, so
C sends R (B,1,C,B) to B

(20) E receives R (B,1,B,E) and is still idle, so
E decrements its $NUM(B)$ by 1
E's $NUM(B)$ is now 1, so E waits for another reply

(21) B receives R (B,1,C,B) and is still idle, so
B decrements its $NUM(B)$ by 1

B's $NUM(B)$ is now 2, so B waits for two more replies

(22) D becomes idle and waits for processes B and E (see Figure 47)

(23) D initiates a query computation
D sends Q (D,1,D,B) to B
D sends Q (D,1,D,E) to E
D sets its $NUM(D)$ to 2

(24) B receives Q (D,1,D,B) and is idle, so
B sends Q (D,1,B,C) to C
B sends Q (D,1,B,D) to D
B sends Q (D,1,B,E) to E
B sets its $NUM(D)$ to 3

(25) E receives Q (D,1,D,E)
E sends Q (D,1,E,B) to B
E sends Q (D,1,E,D) to D
E sets its $NUM(D)$ to 2

(26) C receives Q (D,1,B,C)
C sends Q (D,1,C,B) to B
C sets its $NUM(D)$ to 1

(27) D receives Q (D,1,B,D) and is still idle, so
D sends R (D,1,D,B) to B

(28) E receives Q (D,1,B,E) and is still idle, so
E sends R (D,1,E,B) to B

(29) B receives Q (D,1,E,B) and is still idle, so
B sends R (D,1,B,E) to E

(30) D receives Q (D,1,E,D) and is still idle, so
D sends R (D,1,D,E) to E

(31) B receives Q (D,1,C,B) and is still idle, so
B sends R (D,1,B,C) to C

(32) B receives R (D,1,D,B) and is still idle, so
B decrements its $NUM(D)$ by 1
B's $NUM(D)$ is now 2, so B waits for 2 more replies

(33) B receives R (D,1,E,B) and is still idle, so
B decrements its $NUM(D)$ by 1
B's $NUM(D)$ is now 1, so B waits for 1 more reply

(34) E receives R (D,1,B,E) and is still idle, so
E decrements its $NUM(D)$ by 1
E's $NUM(D)$ is now 1, so E waits for 1 more reply

(35) E receives R (D,1,D,E) and is still idle, so
E decrements its $NUM(D)$ by 1
E's $NUM(D)$ is now 0, so
E sends R (D,1,E,D) to D

(36) C receives R (D,1,B,C) and is still idle, so
C decrements its $NUM(D)$ by 1
C's $NUM(D)$ is now 0, so
C sends R (D,1,C,B) to B

(37) D receives R (D,1,E,D) and is still idle, so
D decrements its $NUM(D)$ by 1
D's $NUM(D)$ is now 1, so D waits for 1 more reply

(38) B receives R (D,1,C,B) and is still idle, so
B decrements its $NUM(D)$ by 1
B's $NUM(D)$ is now 0, so
B sends R (D,1,B,D) to D

(39) D receives R (D,1,B,D) and is still idle, so
D decrements its $NUM(D)$ by 1
D's $NUM(D)$ is now 0, so
D now declares DEADLOCK

Algorithm I

Algorithm I introduces the concept of depth–first querying, cancels, and informs. A query computation terminates when encountering an executing process. The query trace list shows the processes in the reachable set of the initiator of the query computation.

Time Action

(1) A initiates a query computation
A sends Q (A,B,1,A) to B

(2) B receives Q (A,B,1,A) and is idle, so
B sends Q (B,C,1,A,B) to C

(3) C receives Q (B,C,1,A,B) and is idle, so
C sends R (C,B,1,A,B,C) to B

(4) B receives R (C,B,1,A,B,C) and is still idle, so
B sends Q (B,D,1,A,B,C) to D

(5) D receives Q (B,D,1,A,B,C) and is executing, so
D sends I (D,A,1,timestamp) to A
D discards the query

(6) B initiates a query computation
B sends Q (B,C,1,B) to C

(7) C receives Q (B,C,1,B) and is idle, so
C sends R (C,B,1,B,C) to B

(8) B receives R (C,B,1,B,C) and is still idle, so
B sends Q (B,D,1,B,C) to D

(9) D receives Q (B,D,1,B,C) and is executing, so
D sends I (D,B,1,timestamp) to B
D discards the query

(10) D becomes idle and waits for processes B and E (see Figure 47)

(11) D initiates a query computation
D sends Q (D,B,1,D) to B

(12) B receives Q (D,B,1,D) and is idle, so
B sends Q (B,C,1,D,B) to C

(13) C receives Q (B,C,1,D,B) and is idle, so
C sends R (C,B,1,D,B,C) to B

(14) B receives R (C,B,1,D,B,C) and is still idle, so
B sends Q (B,E,1,D,B,C) to E

(15) E receives Q (B,E,1,D,B,C) and is idle, so
E sends Q (E,D,1,D,B,C,E) to D

(16) D receives Q (E,D,1,D,B,C,E) and is still idle, so
D sends R (D,E,1,D,B,C,E) to E

(17) E receives R (D,E,1,D,B,C,E) and is still idle, so
E sends R (E,B,1,D,B,C,E) to B

(18) B receives R (E,B,1,D,B,C,E) and is still idle, so
B sends R (B,D,1,D,B,C,E) to D

(19) D receives R (B,D,1,D,B,C,E) and is still idle, so
D sends Q (D,E,1,D,B,C,E) to E

(20) E receives Q (D,E,1,D,B,C,E) and is still idle, so
E sends R (E,D,1,D,B,C,E) to D

(21) D receives R (E,D,1,D,B,C,E) and is still idle, so
D declares DEADLOCK for processes D, B, C, and E

Algorithm II

Algorithm II introduces the concept of allowing an executing process to hold a query computation and then restart it once the process becomes idle. Each process in the query trace list has its own sequence number.

Time	Action
(1)	A initiates a query computation A sends Q (A,B,t,A,1) to B
(2)	B receives Q (A,B,t,A,1) and is idle, so B sends Q (B,C,t,A,1,B,1) to C
(3)	C receives Q (B,C,t,A,1,B,1) and is idle, so C sends R (C,B,t,A,1,B,1,C,1) to B
(4)	B receives R (C,B,t,A,1,B,1,C,1) and is still idle, so B sends Q (B,D,t,A,1,B,1,C,1) to D
(5)	D receives Q (B,D,t,A,1,B,1,C,1) and is executing, so D sends I (D,A,1,timestamp) to A D holds the query computation
(6)	B initiates a query computation B sends Q (B,C,t,B,1) to C
(7)	C receives Q (B,C,t,B,1) and is idle, so C sends R (C,B,t,B,1,C,1) to B
(8)	B receives R (C,B,t,B,1,C,1) and is still idle, so B sends Q (B,D,t,B,1,C,1) to D
(9)	D receives Q (B,D,t,B,1,C,1) and is executing, so D sends I (D,B,1,timestamp) to B D holds the query computation
(10)	D becomes idle and waits for processes B and E (see Figure 47) D is holding two query computations D sends Q (D,E,t,A,1,B,1,C,1,D,2) to E D sends Q (D,E,t,B,1,C,1,D,2) to E
(11)	E receives Q (D,E,t,A,1,B,1,C,1,D,2) and is idle, so E sends Q (E,B,t,A,1,B,1,C,1,D,2,E,1) to B
(12)	E receives Q (D,E,t,B,1,C,1,D,2) and is idle, so E sends Q (E,B,t,B,1,C,1,D,2,E,1) to B

(13) B receives Q (E,B,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,E,t,A,1,B,1,C,1,D,2,E,1) to E

(14) B receives Q (E,B,t,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,E,t,B,1,C,1,D,2,E,1) to E

(15) E receives R (B,E,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,D,t,A,1,B,1,C,1,D,2,E,1) to D

(16) E receives R (B,E,t,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,D,t,B,1,C,1,D,2,E,1) to D

(17) D receives R (E,D,t,A,1.B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,t,A,1,B,1,C,1,D,2,E,1) to B

(18) D receives R (E,D,t,B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,t,B,1,C,1,D,2,E,1) to B

(19) B receives R (D,B,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends Q (B,E,t,A,1,B,1,C,1,D,2,E,1) to E

(20) B receives R (D,B,t,B,1,C,1,D,2,E,1) and is still idle, so
B sends Q (B,E,t,B,1,C,1,D,2,E,1) to E

(21) E receives Q (B,E,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,B,t,A,1,B,1,C,1,D,2,E,1) to B

(22) E receives Q (B,E,t,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,B,t,B,1,C,1,D,2,E,1) to B

(23) B receives R (E,B,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,A,t,A,1,B,1,C,1,D,2,E,1) to A

(24) B receives R (E,B,t,B,1,C,1,D,2,E,1) and is still idle, so
B declares DEADLOCK for processes B, C, D, and E

(25) A receives R (B,A,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
A declares DEADLOCK for processes A, B, C, D, and E

(26) D initiates a query computation
D sends Q (D,B,t,D,1) to B

(27) B receives Q (D,B,t,D,1) and is idle, so
B sends Q (B,C,t,D,1,B,1) to C

(28) C receives Q (B,C,t,D,1,B,1) and is idle, so
C sends R (C,B,t,D,1,B,1,C,1) to B

(29) B receives R (C,B,t,D,1,B,1,C,1) and is still idle, so
B sends Q (B,E,t,D,1,B,1,C,1) to E

(30) E receives Q (B,E,t,D,1,B,1,C,1) and is idle, so
E sends Q (E,D,t,D,1,B,1,C,1,E,1) to D

(31) D receives Q (E,D,t,D,1,B,1,C,1,E,1) and is still idle, so
D sends R (D,E,t,D,1,B,1,C,1,E,1) to E

(32) E receives R (D,E,t,D,1,B,1,C,1,E,1) and is still idle, so
E sends R (E,B,t,D,1,B,1,C,1,E,1) to B

(33) B receives R (E,B,t,D,1,B,1,C,1,E,1) and is still idle, so
B sends R (B,D,t,D,1,B,1,C,1,E,1) to D

(34) D receives R (B,D,t,D,1,B,1,C,1,E,1) and is still idle, so
D sends Q (D,E,t,D,1,B,1,C,1,E,1) to E

(35) E receives Q (D,E,t,D,1,B,1,C,1,E,1) and is still idle, so
E sends R (E,D,t,D,1,B,1,C,1,E,1) to D

(36) D receives R (E,D,t,D,1,B,1,C,1,E,1) and is still idle, so
D declares DEADLOCK for processes D, B, C, and E

Algorithm III

Algorithm III introduces the concept of a priority scheme to reduce the amount of query traffic. Also, if a deadlocked process receives a query, then it will add all members of its reachable set to the query trace list using a zero sequence number. Just as for Algorithm II, each process in the query trace list has its own sequence number.

Time Action

(1) A initiates a query computation
A sends Q (A,B,1,t,A,1) to B

(2) B receives Q (A,B,1,t,A,1) and is idle, so
B sends Q (B,C,1,t,A,1,B,1) to C

(3) C receives Q (B,C,1,t,A,1,B,1) and is idle, so
C sends R (C,B,1,t,A,1,B,1,C,1) to B

(4) B receives R (C,B,1,t,A,1,B,1,C,1) and is still idle, so
B sends Q (B,D,1,t,A,1,B,1,C,1) to D

(5) D receives Q (B,D,1,t,A,1,B,1,C,1) and is executing, so
D sends I (D,A,1,timestamp) to A
D holds the query computation

(6) B initiates a query computation
B has a higher priority query outstanding, so

B holds Q (B,C,2,t,B,1)

(7) D becomes idle and waits for processes B and E (see Figure 47)
D is holding one query computation
D sends Q (D,E,1,t,A,1,B,1,C,1,D,2) to E

(8) E receives Q (D,E,1,t,A,1,B,1,C,1,D,2) and is idle, so
E sends Q (E,B,1,t,A,1,B,1,C,1,D,2,E,1) to B

(9) B receives Q (E,B,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,E,1,t,A,1,B,1,C,1,D,2,E,1) to E

(10) E receives R (B,E,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,D,1,t,A,1,B,1,C,1,D,2,E,1) to D

(11) D receives R (E,D,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,1,t,A,1,B,1,C,1,D,2,E,1) to B

(12) B receives R (D,B,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends Q (B,E,1,t,A,1,B,1,C,1,D,2,E,1) to E

(13) E receives Q (B,E,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,B,1,t,A,1,B,1,C,1,D,2,E,1) to B

(14) B receives R (E,B,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,A,1,t,A,1,B,1,C,1,D,2,E,1) to A
B sends Q (B,C,2,t,B,1) to C

(15) A receives R (B,A,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
A declares DEADLOCK for processes A, B, C, D, and E

(16) C receives Q (B,C,2,t,B,1) and is idle, so
C sends R (C,B,2,t,B,1,C,1) to B

(17) B receives R (C,B,2,t,B,1,C,1) and is still idle, so
B sends Q (B,D,2,t,B,1,C,1) to D

(18) D receives Q (B,D,2,t,B,1,C,1) and is idle, so
D sends Q (D,E,2,t,B,1,C,1,D,2) to E

(19) E receives Q (D,E,2,t,B,1,C,1,D,2) and is idle, so
E sends Q (E,B,2,t,B,1,C,1,D,2,E,1) to B

(20) B receives Q (E,B,2,t,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,E,2,t,B,1,C,1,D,2,E,1) to E

(21) E receives R (B,E,2,t,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,D,2,t,B,1,C,1,D,2,E,1) to D

(22) D receives R (E,D,2,t,B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,2,t,B,1,C,1,D,2,E,1) to B

(23) B receives R (D,B,2,t,B,1,C,1,D,2,E,1) and is still idle, so
B sends Q (B,E,2,t,B,1,C,1,D,2,E,1) to E

(24) E receives Q (B,E,2,t,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,B,2,t,B,1,C,1,D,2,E,1) to B

(25) B receives R (E,B,2,t,B,1,C,1,D,2,E,1) and is still idle, so
B declares DEADLOCK for processes B, C, D, and E

(26) D initiates a query computation
D sends Q (D,B,4,t,D,1) to B

(27) B receives Q (D,B,4,t,D,1) and is deadlocked, so
B sends R (B,D,4,t,D,1,B,2,C,0,E,0) to D

(28) D receives R (B,D,4,t,D,1,B,2,C,0,E,0) and is still idle, so
D sends Q (D,E,4,t,D,1,B,2,C,0,E,0) to E

(29) E receives Q (D,E,4,t,D,1,B,2,C,0,E,0) and sees its sequence number is 0, so
E sends R (E,D,4,t,D,1,B,2,C,0,E,0) to D

(30) D receives R (E,D,4,t,D,1,B,2,C,0,E,0) and is still idle, so
D declares DEADLOCK for processes D, B, C, and E

Algorithm IV

Algorithm IV introduces the concept of using information obtained from previous query computations. Also, if a deadlocked process receives a query, then it will add all members of its reachable set to the query trace list, but it will not use sequence numbers of 0. With this algorithm, sequence numbers reflect the number of times processes have accepted messages and executed.

Time Action

(1) A initiates a query computation
A sends Q (A,B,1,t,A,1) to B

(2) B receives Q (A,B,1,t,A,1) and is idle, so
B sends Q (B,C,1,t,A,1,B,1) to C

(3) C receives Q (B,C,1,t,A,1,B,1) and is idle, so
C sends R (C,B,1,t,A,1,B,1,C,1) to B

(4) B receives R (C,B,1,t,A,1,B,1,C,1) and is still idle, so
B sends Q (B,D,1,t,A,1,B,1,C,1) to D

(5) D receives Q (B,D,1,t,A,1,B,1,C,1) and is executing, so
D sends I (D,A,1,timestamp) to A
D holds the query computation

(6) B initiates a query computation
B has a higher priority query outstanding, so
B holds Q (B,C,2,t,B,1)

(7) D becomes idle and waits for processes B and E (see Figure 47)
D is holding one query computation
D sends Q (D,E,1,t,A,1,B,1,C,1,D,2) to E

(8) E receives Q (D,E,1,t,A,1,B,1,C,1,D,2) and is idle, so
E sends Q (E,B,1,t,A,1,B,1,C,1,D,2,E,1) to B

(9) B receives Q (E,B,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
B sends R (B,E,1,t,A,1,B,1,C,1,D,2,E,1) to E
B learns that any messages sent to B by E when E was executing with sequence number 1 have arrived.

(10) E receives R (B,E,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
E sends R (E,D,1,t,A,1,B,1,C,1,D,2,E,1) to D

(11) D receives R (E,D,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,1,t,A,1,B,1,C,1,D,2,E,1) to B

(12) B receives R (D,B,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle
B knows all messages sent to it by E when E was executing with sequence number 1 have arrived, so
B sends R (B,A,1,t,A,1,B,1,C,1,D,2,E,1) to A
B was holding a query computation it initiated, so
B sends Q (B,C,2,t,B,1) to C

(13) A receives R (B,A,1,t,A,1,B,1,C,1,D,2,E,1) and is still idle, so
A declares DEADLOCK for processes A, B, C, D, and E

(14) C receives Q (B,C,2,t,B,1) and is idle, so
C sends R (C,B,2,t,B,1,C,1) to B

(15) B receives R (C,B,2,t,B,1,C,1) and is still idle, so
B sends Q (B,D,2,t,B,1,C,1) to D

(16) D receives Q (B,D,2,t,B,1,C,1) and is idle, so
D sends Q (D,E,2,t,B,1,C,1,D,2) to E

(17) E receives Q (D,E,2,t,B,1,C,1,D,2) and is idle
E knows all messages sent to it by B when B was executing with sequence number 1 have arrived, so
E sends R (E,D,2,t,B,1,C,1,D,2,E,1) to D

(18) D receives R (E,D,2,t,B,1,C,1,D,2,E,1) and is still idle, so
D sends R (D,B,2,t,B,1,C,1,D,2,E,1) to B

(19) B receives R (D,B,2,t,B,1,C,1,D,2,E,1) and is still idle, so
B declares DEADLOCK for processes B, C, D, and E

(20) D initiates a query computation
D sends Q (D,B,4,t,D,2) to B

(21) B receives Q (D,B,4,t,D,2) and is deadlocked, so
B sends R (B,D,4,t,D,2,B,1,C,1,E,1) to D

(22) D receives R (B,D,4,t,D,2,B,1,C,1,E,1) and is still idle, so
D declares DEADLOCK for processes D, B, C, and E

Algorithm V

Algorithm V introduces the concept of using information obtained during the current query computation to detect all minimal deadlocked sets. Boolean flags are used to carry this information from process to process, and they will be represented as a sign preceeding the sequence number.

Time Action

(1) A initiates a query computation
A sends Q (A,B,1,t,A,-1) to B

(2) B receives Q (A,B,1,t,A,-1) and is idle, so
B sends Q (B,C,1,t,A,-1,B,-1) to C

(3) C receives Q (B,C,1,t,A,-1,B,-1) and is idle, so
C sends R (C,B,1,t,A,-1,B,-1,C,+1) to B

(4) B receives R (C,B,1,t,A,-1,B,-1,C,+1) and is still idle, so
B sends Q (B,D,1,t,A,-1,B,-1,C,+1) to D

(5) D receives Q (B,D,1,t,A,-1,B,-1,C,+1) and is executing, so
D sends I (D,A,1,timestamp) to A
D holds the query computation

(6) B initiates a query computation
B has a higher priority query outstanding, so
B holds Q (B,C,2,t,B,-1)

(7) D becomes idle and waits for processes B and E (see Figure 47)
D is holding one query computation
D sends Q (D,E,1,t,A,–1,B,–1,C,+1,D,+2) to E

(8) E receives Q (D,E,1,t,A,–1,B,–1,C,+1,D,+2) and is idle, so
E sends Q (E,B,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) to B

(9) B receives Q (E,B,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) and is still idle, so
B sends R (B,E,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) to E
B learns that any messages sent to B by E when E was executing with sequence number 1 have arrived.

(10) E receives R (B,E,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) and is still idle, so
E sends R (E,D,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) to D

(11) D receives R (E,D,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) and is still idle, so
D sends R (D,B,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) to B

(12) B receives R (D,B,1,t,A,–1,B,–1,C,+1,D,+2,E,+1) and is still idle
B knows all messages sent to it by E when E was executing with sequence number 1 have arrived, so
B has finished querying the members of its dependent set.
B has received the final reply with the sign bit for B still FALSE (–).
B knows it is a member of a deadlocked set consisting of B, C, D, and E, so
B declares DEADLOCK for processes B, C, D, and E.
B sends R (B,A,1,t,A,+1,B,–1,C,–1,D,–2,E,+1) to A
The query trace list identifies the minimal deadlocked set to A.
B was holding a query computation it initiated, but this is not needed, so
B discards Q (B,C,2,t,B,–1) as unnecessary.

(13) A receives R (B,A,1,t,A,+1,B,–1,C,–1,D,–2,E,+1) and is still idle, so
A declares DEADLOCK for processes A, B, C, D, and E
A recognizes the minimal deadlocked set consisting of processes B, C, D, and E, and that A is not a member of the minimal deadlocked set.

(14) D initiates a query computation
D sends Q (D,B,4,t,D,–2) to B

(15) B receives Q (D,B,4,t,D,–2) and is deadlocked, so
B sends R (B,D,4,t,D,–2,B,+1,C,+1,E,+1) to D

(16) D receives R (B,D,4,t,D,–2,B,+1,C,+1,E,+1) and is still idle, so
D declares DEADLOCK for processes D, B, C, and E
D recognizes it is a member of a minimal deadlocked set consisting of D, B, C, and E.

Example 2

The following example shows in greater detail how the minimal deadlocked sets are determined by Algorithm V. In this example, suppose that all the processes are idle, and that there are no messages in transit. The system is shown in Figure 48. The query computation initiated by process A is shown.

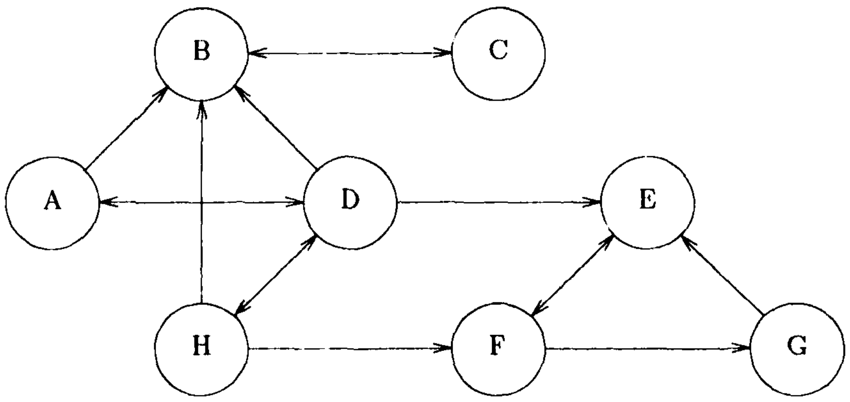


Figure 48
Final Configuration with Deadlock

Time Action

(1) A initiates a query computation
A sends Q (A,B,1,t,A,-1) to B

(2) B receives Q (A,B,1,t,A,-1)
B sends Q (B,C,1,t,A,-1,B,-1) to C

(3) C receives Q (B,C,1,t,A,-1,B,-1)
C sends R (C,B,1,t,A,-1,B,-1,C,+1) to B

(4) B receives R (C,B,1,t,A,-1,B,-1,C,+1)
B recognizes it is a member of a minimal deadlocked set consisting of
B and all processes in the query trace list following B (namely C).
B declares DEADLOCK for processes B and C.
B turns the flag off for A, since A cannot be in the minimal deadlocked set.
B sends R (B,A,1,t,A,+1,B,-1,C,+1) to A

(5) A receives R (B,A,1,t,A,+1,B,-1,C,+1)
A sends Q (A,D,1,t,A,+1,B,-1,C,+1) to D

(6) D receives Q (A,D,1,t,A,+1,B,-1,C,+1)
Since A is in the dependent set of D, D turns off the flags for all processes between A and D in the query trace list.
D sends Q (D,B,1,t,A,+1,B,+1,C,+1,D,+1) to B

(7) B receives Q (D,B,1,t,A,+1,B,+1,C,+1,D,+1)
B sends R (B,D,1,t,A,+1,B,+1,C,+1,D,+1) to D

(8) D receives R (B,D,1,t,A,+1,B,+1,C,+1,D,+1)
D sends Q (D,E,1,t,A,+1,B,+1,C,+1,D,+1) to E

(9) E receives Q (D,E,1,t,A,+1,B,+1,C,+1,D,+1)
E sends Q (E,F,1,t,A,+1,B,+1,C,+1,D,+1,E,-1) to F

(10) F receives Q (E,F,1,t,A,+1,B,+1,C,+1,D,+1,E,-1)
F sends Q (F,G,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1) to G

(11) G receives Q (F,G,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1)
G sends Q (G,E,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1) to E

(12) E receives Q (G,E,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1)
E sends R (E,G,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1) to G

(13) G receives R (E,G,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1)
G sends R (G,F,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1) to F

(14) F receives R (G,F,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1)
F sends R (F,E,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1) to E

(15) E receives R (F,E,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,+1,G,+1)
E recognizes it is a member of a minimal deadlocked set,
along with processes F and G.
E declares DEADLOCK for processes E, F, and G
E changes the flag pattern in the minimal deadlocked set to be (- - - +).
E sends R (E,D,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1) to D

(16) D receives R (E,D,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1)
D sends Q (D,H,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1) to H

(17) H receives Q (D,H,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1)
H sends Q (H,B,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1) to B

(18) B receives Q (H,B,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1)
B sends R (B,H,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1) to H

(19) H receives R (B,H,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1)
H sends Q (H,F,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1) to F

(20) F receives Q (H,F,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1)
F knows from previous queries and replies that the sequence numbers for E and G in the query trace list are the latest ones.
F sends R (F,H,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1) to H

(21) H receives R (F,H,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1)
H sends Q (H,D,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1) to D

(22) D receives R (H,D,1,t,A,+1,B,+1,C,+1,D,+1,E,+1,F,+1,G,+1,H,+1)
D only updates the information about (H,+1) from the query trace list
D sends R (D,A,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1,H,+1) to A

(23) A receives R (B,H,1,t,A,+1,B,+1,C,+1,D,+1,E,-1,F,-1,G,+1,H,+1)
A only updates the information about (D,+1,E,-1,F,-1,G,+1,H,+1) from the query trace list.
A now has a query trace list which shows
(A,+1,B,-1,C,+1,D,+1,E,-1,F,-1,G,+1,H,+1)
A is finished querying its dependent set, so
A declares DEADLOCK for processes A, B, C, D, E, F, G, and H.
A looks for patterns of (– – – – +) in the query trace list to find all the minimal deadlocked sets.
These patterns occur for (B,-1,C,+1) and (E,-1,F,-1,G,+1), which means these form minimal deadlocked sets.
Notice that processes B and E were also able to detect deadlock during this query computation.

VITA

Richard Arthur Adams, the son of Ralph W. Adams and (the late) Constance J. Adams, was born in New Hampton, Iowa on 12 December 1951, and raised on a farm near Waucoma, Iowa. After graduating from Turkey Valley Community School in 1970, he attended Iowa State University in Ames, Iowa, receiving a Bachelor of Science degree in Mathematics and Computer Science and a commission in the USAF through the AFROTC program. He entered active duty as a Second Lieutenant and was stationed at Offutt Air Force Base for three years as a computer systems analyst. From August 1977 to March 1979, he attended the School of Engineering, Air Force Institute of Technology, and received a Master of Science degree in Computer Systems. He then was stationed in Stuttgart, West Germany for four years and while there he earned a Bachelor of Arts degree in German from the University of Maryland, European Division. In June 1983, he entered the University of Illinois to begin graduate studies in Computer Science, and on 1 February 1986 was promoted to his present rank of major. He is a member of the honor societies of Tau Beta Pi, Pi Mu Epsilon, and Phi Kappa Phi. Richard is married to the former Heather M. Olds and they have two children.

END

11-86

DTIC